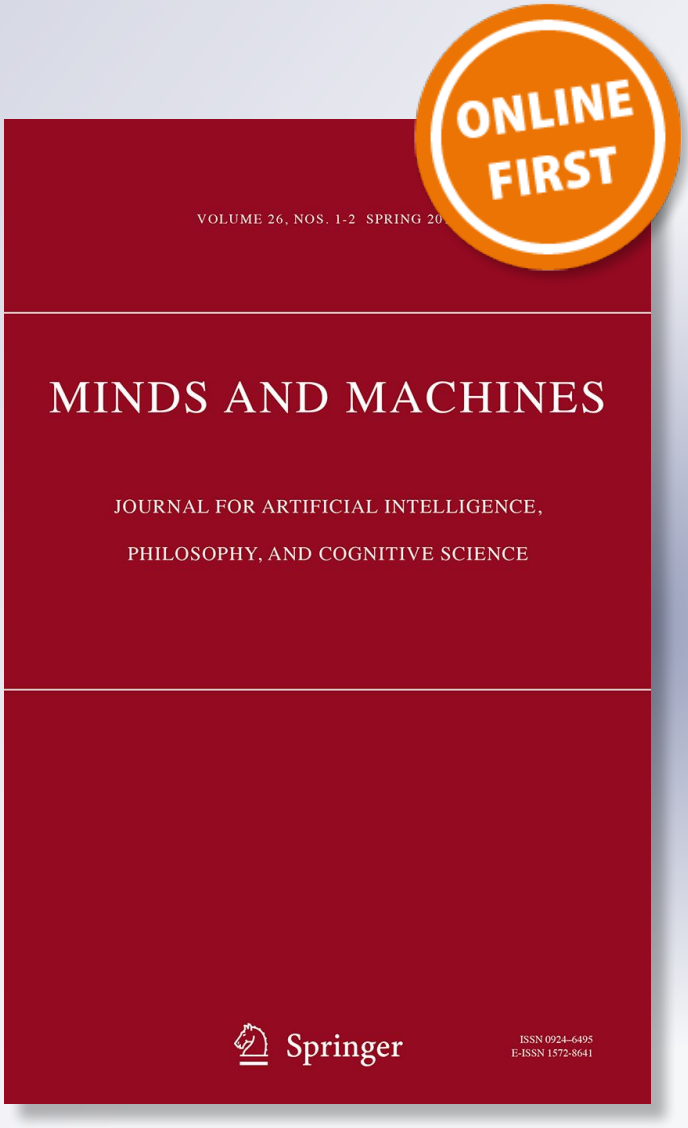
The Bit (and Three Other Abstractions) Define the Borderline Between Hardware and Software

Russ Abbott

Minds and Machines
Journal for Artificial Intelligence, Philosophy and Cognitive Science

ISSN 0924-6495

Minds & Machines
DOI 10.1007/s11023-018-9486-1



Springer

Minds and Machines
https://doi.org/10.1007/s11023-018-9486-1

The Bit (and Three Other Abstractions) Define the Borderline Between Hardware and Software

Russ Abbott[1]

Received: 18 July 2018 / Accepted: 15 December 2018

Abstract
Modern computing is generally taken to consist primarily of symbol manipulation. But symbols are abstract, and computers are physical. How can a physical device manipulate abstract symbols? Neither Church nor Turing considered this question. My answer is that the *bit*, as a hardware-implemented abstract data type, serves as a bridge between materiality and abstraction. Computing also relies on three other primitive—but more straightforward—abstractions: *Sequentiality*, *State*, and *Transition*. These physically-implemented abstractions define the borderline between hardware and software and between physicality and abstraction. At a deeper level, asking how a physical device can interact with abstract symbols is the wrong question. The relationship between symbols and physical devices begins with the realization that human beings already know what it means to manipulate symbols. We build and program computers to do what we understand to be symbol manipulation. To understand what that means, consider a light switch. A light switch doesn't turn a light *on* or *off*. Those are abstractions. Light switches don't operate with abstractions. We build light switches (and their associated circuitry), so that when flipped, the world is changed in such a way that *we* understand the light to be *on* or *off*. Similarly, we build computers to perform operations that we understand as manipulating symbols.

Keywords Symbol · Abstraction · Hardware · Software · Physical symbol system · Hardware–software bridge · Type · Abstract data type · Bit · Affordances · Concept externalization · Symbol grounding

1 Introduction

McCullogh (1960) recounts how Rufus Jones, his undergraduate advisor (and founder of the American Friends Service Committee), asked him,

✉ Russ Abbott
Russ.Abbott@gmail.com

[1] Department of Computer Science, California State University, Los Angeles, Los Angeles, USA

Published online: 01 January 2019

> What is thee going to be? What is thee going to do?
>
> McCullough replied,
>
> I have no idea, but there is one question I would like to answer: What is a number, that a man may know it, and a man, that he may know a number?
>
> Jones said,
>
> Friend, thee will be busy as long as thee lives.

This question, which McCullough (1964) sharpened to "What's in the Brain that Ink May Character?"[1] indeed occupied McCullough throughout his life.

I take McCullough to be asking about the nature of symbols—what are symbols, and how do we as physical beings create and use them.

For example, what do we mean when we say that a symbol that appears multiple times in a mathematical expression (or a logical expression or a computer program) is "the same symbol" at each occurrence? Wetzel (2018) makes this point.

> The variable `x` occurs three times in the formula `∃x(Ax & Bx)`. How can one and the same thing occur more than once?

Is asking this question just playing with words? No one is confused about symbols in logic, mathematics, or software. Is there really a problem?

Before proceeding I'm going to take a liberty with Wetzel's question. She wrote, "The variable `x` occurs three times …" Instead of *variable*, I'm going use the term *symbol*.

> How can the same symbol occur more than once?

One goal of this paper is to answer that question. The larger goal involves exploring the relationship between symbols and computers. It is to investigate how it is even possible for there to be, in the words of Newell and Simon, physical symbol systems. How can physical devices manipulate abstract symbols?

The rest of this paper is organized as follows. Section 2 lays out the problem and describes the four abstractions upon which symbolic computing rests.

[1] "What's in the brain that ink may character" opens Shakespeare's Sonnet 108. McCullough intended that line to mean, "What is a brain that it is able to see pen scratches as characters." Shakespeare's concern is with the opposite journey.

> What's in the brain that ink may character,
> Which hath not figur'd to thee my true spirit?
> What's new to speak, what new to register,
> That may express my love or thy dear merit?

I take the first two lines to mean: What thoughts that may be written down (i.e., that ink may character) about my love for you have not already been expressed?

For both Shakespeare and McCullough, *character* functions (creatively) as a verb—transitive for Shakespeare, intransitive for McCullough—with *ink* (also creatively) as its subject.

Sections 3 and 4 discuss what Church and Turing, respectively, say about symbols and computing. Church (explicitly) took the notion of symbol as primitive; Turing implicitly acknowledged the problem but let it slide.

Section 5 discusses Piccinini's mechanistic computing. He explores the conditions under which physical devices should be credited with performing computations. Piccinini embeds his answer in a fairly complex framework. Despite all his work, he does not confront the issue of how (or even whether) physical mechanisms are able to interact with symbols.

Section 6 discusses Newell and Simon's notion of a physical symbol system. As indicated above, the term itself—*physical symbol system*—highlights the problem, and Newell provides some important insights. In the end, though, the problem is not resolved.

Section 7 follows up on Newell's suggestion that programming language identifiers are closely related to symbols. The fact that programming language identifiers are implemented on physical devices shows that identifiers are (somehow) physically grounded. The section explores their implementation, which seems to lead to an infinite regress.

Section 8 discusses the computer science notion of an abstract data type and explains how abstract data types solve the infinite-regress problem. It also broadens the notion of an abstract data type to include software APIs, user manuals, and functional specifications. Finally, it discusses how computer engineers and software developers split up the work of symbolic computing.

Section 9 discusses affordances, features of an entity that offer agents a means to use that entity for the agent's own purposes. (A wall light switch is a good example.) Affordances enable agents to ground symbols, suggesting that what seems like a magical link between the physical and the abstract may not be so magical after all.

Section 10 offers some final thoughts.

The "Appendix" discusses how the Unicode standard deals with character types.

2 The Problem

This section frames the problem. In particular, it discusses some of the challenges raised by attempts to talk about symbols. It includes discussions of the following topics.

- Difficulties involved with discussing symbols as entities;
- Treating abstraction as mental constructs—rather than as complementary to concrete entities;
- A brief discussion of Montague semantics as an example of how one might talk about mental constructs;
- The ineffability of mental constructs and some implications of that ineffability.

It also explains, briefly, the other computing primitives: sequentiality, states, and transitions.

Finally, it distinguishes symbolic computing from numerical computing and limits the discussion in this paper to the former.

2.1 What are Symbols?

Symbols are central both to how we think and to how computing systems function. Even though we build machines that implement and manipulate them, it's not clear that we can define what we mean by a symbol.

For example, as Wetzel (2018) points out, in mathematics, logic, software, and other symbol-manipulation contexts, we often say that a symbol appears multiple times in a larger construct such as an expression. What do we mean by that? We almost certainly do not mean that some single entity appears multiple times, i.e., that something, for example, jumps from one place to another during, say, our eye saccades.

If, as is far more likely, we are speaking figuratively, we presumably mean that the multiple items we are referring to when we talk loosely about them as "the symbol" all *represent* a single symbol. In that case, can we define a *represents* relation, say `R`, between things that represent symbols and symbols? What might such a relation look like?

- As a binary relation, `R` is a set of ordered pairs. `Domain(R)` consists of elements that represent symbols; `Range(R)` consists of the symbols being represented. But we immediately run into a problem. What do we mean when we say that `Range(R)` consists of symbols? We still haven't defined what we mean by a symbol? Can we outsource that problem to, say, a predicate `S` so that `S(x)` is true or false depending on whether `x` is a symbol? If so, is there a way to generate elements `x` so that `S(x)` is true? None of this seems straightforward. But why is it so difficult?
- Presumably, `R` is a function—at least within a given context. A thing that represents a symbol presumably cannot represent two symbols.[2]
- Given an item in `Domain(R)`, how does one determine which element, if any, in `Range(R)` it represents? That is, how do elements from `Domain(R)` and `Range(R)` get paired up and become a member of `R`? When/how does that happen?
- Given `R(x1, y1)` and `R(x2, y2)`, how do we determine whether `y1` "is the same symbol as" `y2`? That is, is there an `is-the-same-symbol-as` relation on `Range(R)`. Or, perhaps `Range(R)` is a set so that the issue never arises? If so, can such a property be expressed operationally: is there a mechanism that makes a decision about whether an perspective new element of `Range(R)` is already a member?

[2] This ignores scoping issues. ∃x∃y`(Axy & (∀x Bxy))` contains *two* `x` items. The `x` in `Axy` and the `x` in `Bxy` look the same, but they refer to different symbols. (One can see that they refer to two symbols by replacing the two final occurrences of `x` with `z` and noticing that the meaning of the expression is not changed.) To accommodate this broader usage would add complexity but would not change the overall argument. So scope issues are ignored in this paper.

More broadly, does it even make sense to talk about symbols and items that represent symbols in a quasi-mathematical framework like this? If, as I suspect, it does not, is there a better way to talk about them? Although this initial attempt looks like a failure, we will see that it offers a first step toward understanding symbols and things that represent them.

2.2 Abstractions are Mental Constructs

Rosen (2017) begins as follows.

> The abstract/concrete distinction has a curious status in contemporary philosophy. It is widely agreed that the distinction is of fundamental importance. And yet there is no standard account of how it should be drawn. There is a great deal of agreement about how to classify certain paradigm cases. Thus it is universally acknowledged that numbers and the other objects of pure mathematics are abstract (if they exist), whereas rocks and trees and human beings are concrete. Some clear cases of abstracta are classes, propositions, concepts, the letter 'A', and Dante's *Inferno*. Some clear cases of concreta are stars, protons, electromagnetic fields, the chalk tokens of the letter 'A' written on a certain blackboard, and James Joyce's copy of Dante's *Inferno*.
>
> The challenge is to say what underlies this dichotomy, either by defining the terms explicitly, or by embedding them in a theory that makes their connections to other important categories more explicit.

With apologies to Rosen, I believe that the framework just sketched begins the discussion of abstraction by looking at the wrong question. The position I advocate is that the term *abstraction* is essentially a synonym for the term *concept*. Abstractions exist only in the minds of entities whose minds are capable of holding concepts.[3] There is no thought-independent universe of objects that can be partitioned into abstract objects and concrete objects. Every abstraction is a thought—although I'm not in a position to define what I mean by a thought.

In taking this position I am rejecting Frege's position, according to which (as explained by Rosen). abstract objects

> belong to a 'third realm' distinct both from the sensible external world and from the internal world of consciousness.

[3] This statement is too strong. Is a bee's perspective on a flower to which it is attracted an abstraction? When a bee responds to a flower, it cannot be responding to a precise patterns of forms and colors. That is, a slight difference in the data a bee's sense organs receive will not, in most cases, change whether the bee is attracted to the flower. Bees, and other biological organisms, must generalize from the data their sense organs receive to abstractions of some sort—e.g., flower to explore versus flower to ignore. Yet I wouldn't go so far as to claim that all biological organisms that perform this sort of generalization have *concepts* for those abstractions. I would refer to phenomena like this—and similar threshold-like reactions by biological organisms—as what one might call proto-abstractions.

Frege offered a number of arguments against identifying abstractions with thought. One of them asks, for example (again according to Rosen),

> Whose mind contains the number 17? Is there one 17 in your mind and another in mine? In that case, the appearance of a common mathematical subject matter is an illusion.

My answer is that I would not consider the number 17 to be an abstract object. It is a thought and an abstraction but not an abstract object in the sense of an independent entity in a third realm. My perspective is that we learn to think about the number 17 and other mental constructs through education and communication. With practice and interaction we learn how to use concepts consistently. I use the concept I refer to as the number 17 in the same way you do. This is similar to how a red traffic light means stop to those of us raised in a culture that treats red traffic lights as meaning stop.

That it's not controversial to say that we can agree on what a red traffic light means implies that we can take for granted—and can do so without running into serious communication problems—that we already agree about what *red* means.

With this in mind, I find it strange and awkward to call the number 17 an object, even an abstract object—or something that would be an abstract object if it existed. Instead, I prefer to think of abstractions, such as the number 17, as mental constructs, the essence of which are their meanings. In other words, I am rejecting the position, which, according to Rosen, has become widespread in modern English-speaking philosophy, that an appropriate use of the term *abstract* is to refer to entities in Frege's third realm—or even that there is such a third realm.

2.3 Montague Semantics as an Example of How to Talk About Mental Constructs

What, then, do we commonly understand "the number 17" to mean? To answer that question requires a theory of semantics, which is not the subject of this paper. However, I am willing to adopt the position that semantic processes occur in our minds and that we are making progress in explicating those processes. A widely used strategy for that explication involves expressing at least some semantic processes in a version of Montague semantics, which is a form of lambda calculus.[4] Janssen (2017) provides the following brief example.

- The denotation of the determiner *every* is: $\lambda P \lambda Q \forall x [P(x) \rightarrow Q(x)]$. (This is a function of two[5] arguments, P and Q that says Q is true of all things for which P is true.)
- The denotation for the common noun man is a set represented by man.

[4] van Eijck and Unger (2010) and van Eijck (2010) argue that functional programming, using, e.g., Haskell, offers the best platform for computational semantics.

[5] In lambda calculus all functions have a single argument. If one wants a function of multiple arguments, one nests one inside the other ($\lambda P \ \lambda Q \ \ldots$) as in the example of *every*.

- Together, the denotation for the noun phrase *every man* is *every* applied to *man*, which is
- $\lambda P \lambda Q \forall x [P(x) \rightarrow Q(x)](man)$. Through beta-reduction, this can be simplified to $\lambda Q \forall x [man(x) \rightarrow Q(x)]$. (If x is a man, then Q is true of x.) The idea is that one is about to say something, Q, about every man. So *every man* takes Q as an argument.

Under this formulation, the denotation of the abstraction *every man* is a function whose argument is itself a function (or a predicate), i.e., whatever one is about to say about every man.

To adopt some form of Montague semantics is to suggest that every abstraction has a denotation in Montague semantics, essentially a lambda calculus expression. That lambda calculus expression purports to represent the semantics of that abstraction, i.e., to characterize how our mental processes work with an abstraction such as *every man*. Adopting Montague semantics is *not* the claim that our minds are lambda calculus machines, only that lambda calculus lets us come close to charactering how our minds work.

In other words, an abstraction (as a mental construct) has a semantics, which our minds employ when using that abstraction. Montague semantics may or may not be adequate for representing that semantics, but it offers a sense of what such a semantics might be like. Van Eijck and Unger (2010) put it as follows.

> Our ultimate goal is to form an adequate model of parts of our language competence. Adequate means that the model has to be realistic in terms of complexity and learnability. We will not be so ambitious as to claim that our account mirrors real cognitive processes, but what we do claim is that our account imposes constraints on what the real cognitive processes can look like.

Given this perspective, it no longer makes sense to talk about a universe of objects to be partitioned into those that are abstract and those that are concrete. We are now saying that abstractions are semantic constructs, something like expressions in the lambda calculus, whose meanings inhere in their evaluation. Separately one may say that concrete objects are things like elements of the physical world. These two categories, lambda calculus expressions and material objects, have little to do with each other. Treating them as a fundamental partitioning of a universe of objects strikes me as sending the discussion of *abstraction* off in an unproductive direction.

2.4 Abstractions are Ineffable

If one accepts the proposition that abstractions are mental constructs it seems to follow immediately that they are ineffable, i.e., that they cannot be displayed or exhibited, e.g., in an academic paper. The best one might do is to express in some language—such as the lambda calculus—the mental processes involved. But a lambda calculus expression is not the same as the process of applying that lambda calculus expression to an argument and evaluating the result. Even a discussion of how one evaluates a lambda calculus expression, e.g., via substitution, etc., is not the same as

the actual process of doing the evaluation. This is similar to saying that the execution of a computer program is not the same as either (a) the program itself or (b) a characterization of how the execution of the program works, i.e., a characterization of the formal semantics of the program. In other words, the process of thinking—or of executing a computer program—cannot (yet?) be captured and offered up as a fixed "thing" to be examined.

So what does one do if one is responsible for managing abstractions? The following examples illustrate how two very different sorts of abstractions are handled by organizations charged with their care.

- The US Copyright Office (2017, FAQ) lists two criteria for copyright eligibility: an item must be both an original work of authorship and fixed in a tangible medium of expression. To apply for copyright, one must present to the copyright office a concrete expression of the item for which protection is sought. For example, a novel may be copyright protected, but to apply for that protection one must submit a concrete expression of that novel. Yet it is not the concrete expression that is protected. When a concrete expression of a novel is submitted for protection, that concrete expression will have properties, such as font characteristics, that are not properties of the protected entity, and hence not protected.

  Why not require that the item itself for which protection is sought be submitted? It seems to me that the simple answer is that the item to be protected is a type, and like all abstractions, a type cannot be exhibited, displayed, etc.,[6] and hence cannot be submitted as part of a copyright application.

- The Unicode Standard (Unicode 2017) provides a way to refer to any character in any of the world's written languages. Here is how it describes itself.

  > The Unicode Standard is the universal character encoding standard for written characters and text. It defines a consistent way of encoding multilingual text that enables the exchange of text data internationally and creates the foundation for global software.

  In the Standard, characters—e.g., digits, letters of alphabets, even emojis—are abstract types. (As Rosen says, the letter 'A' is an abstraction.) The Standard does not exhibit or display any of them: it names them, but it doesn't display them. Why not? Because, I claim, like other abstractions, character types cannot be exhibited or displayed.

  The Standard does include what it calls character *glyphs*, i.e., ways to display characters. But a character may have many glyphs, none of which is the character (type) itself. The "Appendix" explains in more detail how it is possible to create a world-wide and nearly universally accepted standard for (abstract) characters even though the (abstract) characters themselves cannot be exhibited or displayed.

[6] See Wetzel (2018) for additional discussion.

In these two examples we see two well-established organizations whose primary function is to work with abstractions. Yet in both cases they neither display nor exhibit the abstractions with which they work. Furthermore, they do not explain why they do not display those abstractions. They simply don't. This strikes me as fairly strong, although obviously not conclusive, evidence that abstractions simply cannot be displayed or exhibited.

Yet in speaking about abstractions we rarely, if ever, acknowledge that they cannot be displayed or exhibited. (Again, think of the letter 'A'.) I suggest that the reason we don't think about abstractions as ineffable is because they feel so familiar. We think with them and about them all the time. They may even feel visible because in some sense they appear in our minds.

Consider the following thought experiment. Look inside a computer as it is computing. What would one see? A naïve answer is that one would see symbols being manipulated. But symbols are abstractions, and as such are not visible.

What about bits being pushed around? The bit is taken as the iconic, foundational computing element. If one looked inside a computer as it is computing would one see bits being manipulated? Again, the answer is no. Bits, too, are abstractions—see Sect. 8 for how bits as abstractions can be defined and created. We would not see bits being manipulated within a computer as it is computing. What would we see? Piccinini's mechanistic computing, discussed in Sect. 5, looks into that question.

2.5 There is More to Computing than Symbols

Besides symbols, every model of computing, at least every model of which I am aware, rests on three additional abstractions: structures, states, and transitions. Unlike symbols, these abstractions seem directly intuitive and unproblematic. I mention them because (a) they are abstractions, (b) they are primitive (models of computing do not implement them in terms of anything more primitive), and (c) they are fundamental to computing—all models of computing rely on them. This section clarifies what I mean by these terms.

Symbol structures. Computing requires the ability to combine symbols into (mutable or immutable) structures and to modify those structures (when they are mutable), to create variants of them (when they are immutable), and to combine them into additional structures. The most widely adopted structure is the sequence. The Turing machine tape is a prototypical example. The primary alternative is addressable memory. Both are mutable. Symbol structures are abstractions; their properties are both intuitively straightforward and generally taken as primitive.

Although mutability is more common, it is not strictly required. Functional programming languages, such as Haskell, use immutable symbol structures. The functional programming analog to changing some part of a symbol structure is to construct a new symbol structure that is the same as the original except where the original is intended to be modified.

The mutability-immutability contrast becomes clear when we compare the Turing machine to the lambda calculus. The Turing machine uses a mutable symbol structure, which is recorded on the Turing machine tape. The lambda calculus uses

immutable symbol structures. A new symbol structure is created when one wants to represent a change to an existing one. To accomplish this trick, functional programming, including the lambda calculus, assumes the ability to conjure up symbol structures as needed. There is no discussion of the "stuff" of which sequences are built. One assumes they can be made to materialize—in some abstract sense—at will. We do the same thing, of course, by adopting the notion of an unbounded Turing machine tape. Additional tape squares may be summoned up as needed.

One can, in fact convert the Turing machine model into one that includes an immutable tape—immutable in the sense that once the machine writes on a square, that square never changes. Wang (1957) showed that non-erasing Turning machines are universal. Either way, computing simply assumes a basic level of structure—typically sequentiality.

State Modern digital computing relies on the notion of state. To say that a computation assumes states is to say that it is possible at regular intervals to describe the static structure of a computation as a particular organization of a finite number of symbols.

State transitions Computations progress from state to state. The step from one state to the next is generally taken as atomic. High level transitions may be defined in terms of lower level transitions, but there are always primitive atomic transitions that are simply assumed. Earlier we discussed the impossibility of displaying the process of computing. One can display states, which act like milestones (or snapshots) along the path a computing process takes. But the individual atomic transitions? By definition, since they are atomic, they cannot be broken down into something simpler.

Real-life physical computers make these three additional abstractions available. This is part of how hardware performs the magic that transforms the physical into the abstract.

2.6 Bits Versus Quasi-Real Numbers

Computing has developed along two paths: traditional computer science and what has come to be known as computational science. According to *Nature*[7]:

> Computational science is a discipline concerned with the design, implementation, and use of mathematical models to analyze and solve scientific problems. Typically, the term refers to the use of computers to perform simulations or numerical analysis of a scientific system or process.

Whereas the fundamental computer science element is the bit, the fundamental computational science elements are the real (and perhaps complex) numbers. Computational science software computes with real number approximations and uses all

[7] *Nature* has identified "Computational Science" as one of its "subject areas." See https://www.nature.com/subjects/computational-science. See also Humphreys (2004) for an extended discussion and review.

the precision it is given. These real number approximations serve as stand-ins for real numbers—not as stand-ins for symbols.

Blum et al. (1989) models computing of this sort as a Turing machine whose squares hold actual real numbers—recall that this is a theoretical model—rather than discrete symbols. These are now called BSS-machines for the authors of that paper.

This paper does not consider issues arising in computing of that sort.

3 Lambda Calculus and the Term *Occurrence*

In remarks preceding his definition of the lambda calculus, Church (1932) declares a number of concepts as primitive. [Emphasis in the original.]

> We assume that we know the meaning of the words symbol and formula (by the word formula we mean a set of symbols arranged in an order of succession, one after the other). We assume the ability to write symbols and to arrange them in a certain order on a page, and the ability to recognize different occurrences of *the same* symbol and to distinguish between such a double occurrence of a symbol and the occurrence of *distinct* symbols. …
>
> We assume that we know what it means to say that a certain symbol or formula *occurs* in a given formula. …
>
> We assume an understanding of the operation of *substituting* a given symbol or formula *for a particular occurrence* of a given symbol or formula. (p. 350)

In other words, Church assumes that we know what a symbol is, what a symbol occurrence is, and how the basic symbol manipulations are performed. These are taken as primitive. Church doesn't address the fundamental issue of this paper: the relationship between symbols and physical computing devices. Nor does he address the questions raised in Sect. 2.1 regarding the relationship between symbols and their occurrences.

As the extract above indicates, Church requires a number of fundamental symbol properties: distinguishability, occurrence recognition, the ability to arrange symbol occurrences into expressions, and the ability to transform expressions by substituting symbol occurrences or expressions for other symbol occurrences.

Church also assumes the other abstractions: sequences (Church's formulae), states (a partially evaluated expression), and transitions (the step from one state to the next by substitution, i.e., beta-reduction).

In short, Church is clear that he needs the abstractions discussed above. He makes use of them throughout. He takes them as primitive, i.e., as built into the mathematical framework he uses to formulate the lambda calculus.

The term *occurrence* deserves a bit more attention. In software (or mathematical or logical expressions), one almost never deals with physical entities. One does not, for example, talk about ink marks at particular positions on paper. For the most part, one works with (abstract) symbols and (abstract) expressions composed of (abstract)

symbols. As Wetzel (2009) points out, unless one is talking about a particular concrete representation of an expression, e.g., on a whiteboard, *expressions themselves are types*. As such, expressions are abstractions and cannot include physical tokens.

This raises the question of what term to use when referring to what would seem to be the appearance of a symbol in multiple places in an expression. If the expression were concrete, *token* would be the appropriate term. But expressions are not concrete.

The problem occurs whenever a type has other types within it. Wetzel offers the following example.

> What could it mean to say that the very same atomic (type), hydrogen, "occurs four times" in the methane molecule? We are not talking about a particular methane molecule, but the chemical compound methane, i.e., methane as a type. How can the very same thing "occur" more than once? – Paraphrased from Wetzel (2018).

As we saw above, Church used the term *occurrence* for this situation. He did so without drawing attention to the term as in any way special. Look again at the following portion of this extract from Church. The term occurrence appears three times, seemingly as an unremarkable English word.

> … the ability to recognize different occurrences of the same symbol and to distinguish between such a double occurrence of a symbol and the occurrence of distinct symbols.

Wetzel (2018) attributes this usage to logicians in general.

For the purposes of this paper, it is important to have a term for the appearance of a symbol in an abstract context. Since *occurrence* serves this purpose for logicians—and since I have not been able to come up with a better term—I will adopt it also. Thus, the expression `∃x(Ax & Bx)` includes three occurrences of the symbol `x`.

We can now revisit relationship `R,` considered in Sect. 2.1, between things that represent symbols and symbols. If there were such a relationship, its `Domain` would be symbol occurrences. Unfortunately, this doesn't solve the difficulties that `R` presents. As we will see in Sect. 7, symbol tables, although not the same as `R`, perform a similar function.

4 Turing Machines

The following is Turing's (1936) informal description of his machine. (Quotation marks in the original.)

> We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions q_1, q_2, …, q_l, which will be called "*m*-configurations". 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". At any moment there

> is just one square, say the *r*-th, bearing the symbol G(*r*) which is "in the machine". We may call this square the "scanned square". The symbol on the scanned square may be called the "scanned symbol". The "scanned symbol" is the only one of which the machine is, so to speak, "directly aware". However, by altering its m-configuration the machine can effectively remember some of the symbols which it has "seen" (scanned) previously. The possible behaviour of the machine at any moment is determined by the *m*-configuration q_n and the scanned symbol G(*r*). … 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 the right or left. (p. 231)

As Turing writes, the machine operates with a "tape," divided into "squares," each capable of bearing a "symbol." The terms in quotation marks are not further defined and are treated as primitive. (Other terms in quotation marks, such as "directly aware" and "seen" are presumably intended more informally.)

The operation of Turing machines is governed by a set of transition rules. Each transition rule consists of a condition part and an action part. The condition part indicates the conditions under which the rule applies. The action part indicates a transition to be performed.

The condition part consists of a machine configuration, now more frequently called a state, and a symbol, the symbol in the square under the read head. The rule applies when the machine is in the given state and the read head is scanning the given symbol.

There are many equivalent ways to specify the action part. In Turing's formulation the action part consists of (a) an action—write a specified symbol to the currently scanned square or move the read/write head one square to the left or right—and (b) a configuration (i.e., state) which the machine is to assume after the action is performed.

Symbol occurrences may appear in both the condition part and the action part. Such symbol occurrences are assumed to be limited to a pre-specified finite set of symbols. Turing does not discuss what he means by a symbol in this context. Presumably he uses the notion of a symbol the same way it is used elsewhere in mathematics, i.e., it is not defined. (Church was unusual in stating explicitly that he was assuming the notion of symbol.)

Even though Turing introduces his machine as something that could be physical, he makes no pretense that the transition rules are physical. His analogy is that the transition rules are in the minds of people doing computations. They are concepts, i.e., abstractions, uncorrupted by the sort of wear-and-tear or frayed edges associated with concrete machinery.

If the symbols in the transition rules are mathematical abstractions, how should we understand symbols on the tape?

Turing wanted his tape to be concrete. In Sect. 9, he talks explicitly about the physical representation of symbols.

> If we regard a symbol as literally printed on a square, [the] symbol is defined as [the] set of points … occupied by printer's ink. (p. 249, footnote)

In other words, the symbols on the tape are symbol tokens rather than abstractions.[8] This raises the question of how the machine translates between the tokens on the tape and the abstractions in the rules. Turing does not discuss this issue. His machine is intended as a mechanized version of a human being carrying out a computation. A human being would have no trouble translating between symbol tokens and abstract symbols. We do it all the time. Turing may simply have ignored this issue because humans do it so effortlessly.

Although it may seem effortless (perhaps because we have forgotten how difficult it was to learn to read), I think it's fair to say that we do not understand how humans do it. This is not to say that computing systems could not be developed to do something similar. In fact, modern OCR devices, and more impressively deep neural nets, are quite capable of *naming* the abstractions represented by patterns of printer's ink—or of pixels.[9] So I'm willing to grant the possibility that a machine is capable of translating between symbol tokens and names for the symbol abstractions they instantiate. (This is discussed in more detail below.)

Turing does not say any more about what the symbol abstractions themselves are. Nor does he discuss the issue of going in the reverse direction: creating symbol tokens to be written on the tape.[10]

After the informal machine description extracted above, Turing writes, [emphasis added]

> In the next section I … proceed with the development of the theory [of these machines] and *assume that it is understood what is meant by 'machine', 'tape', 'scanned',* etc. (p. 232) [The *etc.* presumably includes symbols.]

Where does this leave us? It leaves us with a Turing machine, one of whose important functions is to translate back and forth between physical symbol tokens on a physical tape and abstract symbol occurrences in its transition rules. As I said, I grant that it is possible to translate from symbol tokens to symbol names. But we are left with the conclusion that Turing ignored the token-to-symbol-name translation function. It also leaves as primitive *state*, *tape* (as a sequence of "squares"), and *transition* (in the sense of what a rule's action part means).

In the end, Turing, like Church, takes the notions of an abstract symbol and the other basic abstractions as primitive. Of course this doesn't diminish Turing's

[8] Here, *token* is meant in the conventional way, i.e., as a concrete instance of the symbol as a type. Below we see how Newell uses the notion of a symbol token more abstractly.

[9] It's also possible to fool image recognition systems with what are called adversarial images. These are images that a person would confidently identify as one thing but that the best AI image recognizers would confidently identify as another. The field is developing quickly enough that rather than give a specific reference I suggest a web search for *adversarial images*.

[10] This may seem trivial since humans do it so easily. But recall the discussion of the Unicode Standard. It doesn't display letter abstractions; it doesn't adopt particular glyphs as letter tokens; and it doesn't discuss the issue of how to translate back and forth between letter abstractions and letter glyph tokens. These non-trivial issues are beyond the scope of the Standard.

achievement, which would be just as important had he defined his machine purely mathematically—as Church defined lambda calculus. Turing's lack of attention to the issue of translating between tokens and symbol-names simply confirms that the fundamental models for computing were not designed to clarify the nature of symbols and the other computing abstractions.

Turing requires the same properties and capabilities of symbols as does Church: distinguishability, occurrence recognition, the ability to arrange symbols into sequences (by using a "tape"), the ability to change these sequences (by writing a symbol token on or erasing a symbol token from a tape square), the notion of the state of a machine, and the notion of atomic transitions from one state to another.

5 Mechanistic Computing

Piccinini (2015)[11] proposes to establish a set of conditions under which one can say that a concrete physical system performs a computation—where *to perform a computation* is defined to mean: to implement a formally defined abstract process that is taken as being computation in the fields of computer science and computability theory.

As he puts it,

> The same entities studied in the mathematical theory of computation—Turing machines, algorithms, and so on—are said to be implemented by concrete physical systems. This poses a problem: under what conditions does a concrete, physical system perform a computation when computation is defined by an abstract mathematical formalism? This may be called the *problem of computational implementation*. (p. 6).

We already know how to build physical computers. No one argues that what we call computers implement formally defined abstract processes that are taken as being computation in the fields of computer science and computability theory.

It would seem then that the important question for Piccinini is how to define criteria that exclude physical processes from being considered implementations of computation. Indeed, chapter 1, Sect. 4 lays out what strike me as useful desiderata for any science of concrete computation. Chapters 2, 3, and 4 then argue that certain approaches to physical computation, such as pancomputationalism, do not satisfy one or more of these criteria. So the negative task is done.

Since we already know how to build physical computers, what's left to do? Much of the rest of the book describes (often in fairly abstract terms) physical structures and processes that Piccinini seems to imply *every* case of physical computing must include. In other words, Piccinini lays out not just desiderata but a constructive framework that he believes *every* physical computing system must incorporate.

[11] Cuffaro and Fletcher (2018) collects a dozen contributions to the field of physical computing—including one by Piccinini and Anderson. In this paper, I focus on Piccinini because his work is the most completely fleshed out.

Considering the universal quantifier, making such a case strikes me as quite challenging. And to be fair, Piccinini doesn't argue for it explicitly. But given that we already have physical computers, that would seem to be the task he has laid out for himself. In describing the required structures and processes, Piccinini is forced to address our issue: how can a physical device manipulate abstract symbols. This section examines his discussion in some depth.

5.1 Digital Computing

Piccinini argues that his approach applies to what he calls generic computing, which includes digital computing, analog computing, neural nets, and more. (p. 124) Since in this paper I am concerned with symbols and symbol manipulation, which occur primarily in digital computing, I will focus on Piccinini's approach to digital computing.

Piccinini characterizes digital computation (paraphrased from pp. 125–126) in terms of the following.

(a) *A finite alphabet of letters*. A letter is a type of entity that is distinct from other letter types. These letters are the *computational data*. The alphabet includes a distinguished letter known as the *blank* letter. Piccinini refers to individual letters as types. Although Piccinini does not adopt the term, each appearance of a letter will be referred to as an *occurrence* as discussed in Sect. 3.
(b) *A second and disjoint finite alphabet*, each of whose letters represents one of the possible *states* of the mechanism that is performing the computation. These may also be understood in terms of types and occurrences.
(c) *Strings*, which are ordered sequences of letter occurrences from these two alphabets. A string is individuated by the types of letter that compose it, their number, and their order within the string.
(d) *A list of instructions*, called a program, for generating new strings from old strings.

Digital computations are defined over strings that consists of computational data along with a single letter occurrence from the state alphabet. Strings of this form are often called snapshots.

A digital computation is a sequence of snapshots such that each member of the sequence derives from its predecessor by some instruction in the program. In each snapshot, the sequence of computational data letter occurrences along with the identity and position of the state letter occurrence characterizes the overall state of the computation.

Readers may recognize the preceding as one of the standard representations for Turing machine computations: the state letter identifies the Turing machine state; the computational data letters indicate the contents of the Turing machine tape; the position in the string of the state letter indicates the position of the Turing machine's read/write head. The Turing machine may be understood as scanning the data letter immediately following the state letter. If the state letter occurrence is at the end

of the string, an occurrence of the distinguished *blank* data letter is automatically added to the string after it.

5.2 Implementation and Physical Computing

The preceding framework assumes the existence of symbols: the letter types. The question currently under investigation is how physical devices are able to manipulate symbols. Piccinini's answer will depend on how he reifies letter occurrences and strings of letter occurrences.

His approach, which we discuss in more detail later in the section, involves what he calls digits, which he refers to as medium-independent vehicles, and strings of digits. Here is an overview.

Piccinini uses the term *digit* in multiple ways. *Digit* may refer to (in the following, letter is understood as in the preceding section):

(a) a letter (as a type),
(b) a letter occurrence,
(c) a placeholder for any letter in the alphabet, i.e., a variable whose values range over letter occurrences in the alphabet, and
(d) a physical mechanism capable of bearing a letter occurrence.

A *vehicle* is something like a physical mechanism capable of bearing a value. Every digit is a vehicle. A *medium-independent vehicle* is a vehicle with the property that the value it bears is independent of the physical mechanisms it uses to bear that value.

Given this terminology, Piccinini discusses what it means for a physical device to implement an abstractly defined computational process.

> [T]he same formal operations and rules that define mathematically defined computations over strings of letters can be used to characterize concrete computations over strings of digits. Within a concrete digital computing mechanism, the components are connected so that the inputs from the environment, together with the digits currently stored in memory, are processed by the processing components in accordance with a set of instructions. During each time interval, the processing components transform the previous memory state (and possibly, external input) in a way that corresponds to the transformation of each snapshot into its successor. The received input and the initial memory state implement the initial string of a mathematically defined digital computation. The intermediate memory states implement the intermediate strings. The output returned by the mechanism, together with the final memory state, implement the final string. …
>
> [A]ny system whose function is generating output strings from input strings (and possibly internal states), in accordance with a general rule that applies to all strings and depends on the input strings (and possibly internal states)

for its application, is a digital computing mechanism. (paraphrased from pp. 132–134)

In short, the process the physical device performs must essentially mirror the process the mathematical abstraction specifies. This seems quite rigid. But it serves our purpose well. Since we are interested in how a physical device can manipulate symbols, and since Piccinini's definition of digital computing involves the manipulation of symbols, the rigidity of this definition will require that Piccinini solve the problem of how a physical device can manipulate abstract symbols.

Piccinini sums up his definition of physical computing as follows.

> A physical computing system is a mechanism whose teleological function is performing a physical computation. A physical computation is the manipulation (by a functional mechanism) of a medium-independent vehicle according to a rule. A medium-independent vehicle is a physical variable defined solely in terms of its degrees of freedom (e.g., whether its value is 1 or 0 during a given time interval), as opposed to its specific physical composition (e.g., whether it's a voltage and what voltage values correspond to 1 or 0 during a given time interval). A rule is a mapping from inputs and/or internal states to internal states and/or outputs. (p. 10)[12]

The remainder of this section discusses whether, in my view, Piccinini approach answers the question of how physical devices can manipulate abstract symbols.

5.3 Does it Make Sense to Constrain Vehicles in Terms of Their Degrees of Freedom?

A key term in the preceding is *medium-independent vehicle*, which is characterized as "a physical variable defined solely in terms of its degrees of freedom." It's not clear to me how a variable can be defined in terms of its degrees of freedom. A system's degrees of freedom are the independent dimensions along which it can vary. For example, a point in a 2-dimensional plane has two degrees of freedom. What can it mean for a variable—at least one whose value is not an encoding of multiple independent values[13]—to have more than one degree of freedom?

It makes more sense to me to say that the variable must be defined solely in terms of the values it can assume. For example, a binary variable may assume two values. If one needs a variable that can assume three values, a binary variable will not do. But this is not a matter of degrees of freedom.

There is room for uncertainty, though. Piccinini says that a vehicle

> can be used to refer to one of two things: either a variable, that is, a state that can take different values and change over time, or a specific value of such a

[12] Several similar definitions are provided throughout the book.

[13] A variable whose possible values range over the complex numbers does, indeed, have two degrees of freedom, but most variables are not of this form.

> variable. … [For example, a] string of digits … is a kind of vehicle that is made out of digits concatenated together. (p. 121)

As a vehicle, a string has as many degrees of freedom as it has letters. But that typically is not a crucial issue. When a string is used to represent a Turing machine state, the number of letters it contains grows as the tape adds squares.

5.4 What is a Digit?

A digit is intended as something like a physical version of a letter. In particular,

> A digit is a portion of a vehicle that can take one out of finitely many states during a finite time interval. Typically, a digit is a binary vehicle; it can take only one out of two states. (p. 121)

Piccinini (deliberately) uses the term *digit* ambiguously, to refer either to

> a variable, that is, a state that can take different values and change over time, or a specific value of such a variable. (p. 121)

This ambiguity, he says, "is harmless because context will make clear what is meant in any given case." (both p. 121 and p. 127)

Piccinini uses *digit* most frequently to refer to a concrete counterpart of a letter occurrence rather than to a variable. An important issue, then, is to clarify what *digit* means in that sense.

> To show how a concrete mechanism can perform digital computations, the first step is finding a concrete counterpart to the formal notion of letter from a finite alphabet. I call such an entity a digit. The term 'digit' may … denote the specific state of a variable—e.g., a switch in the off position [or] a memory cell storing a '1'. (p. 127)

This leads to an ambiguity that Piccinini does not address. A digit may be a concrete physical condition, such as a switch being on or off, as immediately above. To be fair, the extract says that a digit may *denote* a switch being on or off. It doesn't say that the switch itself is a digit.[14] But in that case a digit is symbolic, i.e., an abstract entity that has a denotation. But Piccinini never explains how a physical system converts a physical state, such as a switch being on or off, into a symbol that denotes that state. Nor does he says what mechanism interprets the digit as having a denotation and then treats it in accordance with its denotation. Finally, he does not explain what a digit is in physical terms and how it carries its denotation. Presumably a digit is physical since we are talking about a physical system, but it is also symbolic since it has a denotation. Piccinini does not explain what sort of thing can be both.

[14] Is it fair to say that a switch is a vehicle?

The only sense I can make of the extract is to take the switch itself as a digit. But in that case, a digit is not an occurrence of a letter (or other abstract value). It is a physical token of such a letter (or value)[15] type.

5.5 An Unacknowledged Ambiguity

In addition to the use of *digit* as referring to a letter token (or a property of a physical object as above), Piccinini also uses *digit* to refer to both (a) a letter occurrence as well as to (b) the letter type itself.

> A digit may be transmitted as a signal through an appropriate physical medium. It may be transmitted from one component to another of a mechanism. It may also be transmitted from outside to inside a mechanism and vice versa. So a digit may enter the mechanism, be processed or transformed by the mechanism, and exit the mechanism (to be transmitted, perhaps, to another mechanism). (p. 127)

In this case the ambiguity is not harmless; it hides a significant problem. In the preceding, *digit* can be understood meaningfully only as referring to a letter type and not to any of its tokens or occurrences.

In particular, at the physical level it doesn't make sense to say that a switch being on or off is itself transmitted in any of the ways mentioned above. A switch being on or off is a physical property of a particular physical object. That physical property of that particular physical object cannot be transmitted from component to component. The switch itself is fixed in place within the concrete computational mechanism and is not transmitted (in its on or off state) from one component to another, and it's not clear what it would mean for a property of that physical object, i.e., being on or off, to be transmitted. What one would like to transmit is something like the abstract letter type represented by the state of the switch. But, of course, since letter types are abstract, they cannot be transmitted as such.

5.6 Can There Be Medium-Independent Physical Values

Piccinini's notion of medium-independence seems like an attempt to get around this problem.

> [A] vehicle is medium-independent just in case the rule that [manipulates it] is sensitive only to differences between portions (i.e., spatiotemporal parts) of the vehicles along specific dimensions of variation. (p. 122)

So, medium-independence is not a property of vehicles. A vehicle is medium-independent when the rules that manipulate it make it medium independent.

[15] Recall that computational data consists of "letters" of an arbitrary finite alphabet. These letters may represent such discrete values as *on*/*off*, *0*/*1*, etc. and need not be understood in the context of elements of text.

> A rule in the present sense is a map from inputs (and possibly internal states) to outputs. … When we define concrete computations and the vehicles that they manipulate, we need not consider all of their specific physical properties. We may consider only the properties that are relevant to the computation, according to the rules that define the computation. (p. 121)

Piccinini does not discuss what such rules are like. What sort of rule in a physical context can process a digit, such as a switch being on or off, without having to determine whether the switch is actually on or off? What is the non-concrete property that such a concrete switch affords a rule that manipulates concrete digits so that the rule can function in a medium-independent manner? How is such a property communicated from switch to rule? What is the physical mechanism that such a rule employs so that it can avail itself of that affordance?

Piccinini does not explore this issue and concludes as follows.

> Since concrete computations and their vehicles can be defined independently of the physical media that implement them, I call them medium-independent. That is, computational descriptions of concrete physical systems are sufficiently abstract as to be medium-independent. (p. 122)

5.7 What are The Real Constraints?

Piccinini recognizes that this is difficult territory. For example, in talking about logic gates he writes as follows.

> In a digital computing mechanism, under normal conditions, digits of the same type[16] affect primitive components of a mechanism in sufficiently similar ways that their dissimilarities make no difference to the resulting output. For instance, if two inputs to a NOT gate are sufficiently close to a certain voltage (labeled type '0'), the outputs from the gate in response to the two inputs must be of voltages different from the input voltages but sufficiently close to a certain other value (labeled type '1') that their difference does not affect further processing by other logic gates. (p. 129)

Elsewhere, Piccinini attempts to clarify what "sufficiently close" must mean.

> Primitive computing components, such as logic gates, can be wired together to form computing mechanisms, whose computations can be logically analyzed into the operations performed by their components. But not every collection of entities, even if they may be described as logic gates when they are taken in isolation, can be connected together to form a computing mechanism. For that to happen, each putative logic gate must take inputs and generate outputs of the same kind, so that outputs from one gate can be transmitted as inputs to other gates. In addition, even having components of the right kind is not enough to

[16] Here, we are talking about digits as letter or value tokens. "Of the same type" means something like having the same value.

> build a complex computing component. For the components must be appropriately organized. The different gates must be connected together appropriately, provided with a source of energy, and synchronized. To turn a collection of logic gates into a functioning computer takes an enormous amount of regimentation. (p. 148)

To provide this regimentation, Piccinini's strategy is to map equivalence classes of what he calls (physical) microstates of vehicles (e.g., voltage levels) to digit types. A primary concern then becomes to define the mapping in such a way that even though the physical mechanism is actually manipulating microstates, the operation of the mechanism can be understood as manipulating digit types.

This is a fundamental issue, and as Piccinini notes,

> a typical physical variable—including the variables to be found inside digital computers—may vary continuously and take (or be assumed to take) one out of an uncountable number of states at any given instant. And even when a physical variable may take only finitely many states, the number of physical microstates is many orders of magnitude larger than the number of digits in a computing system. Therefore, many physically different microstates count as digits of the same type.
>
> [Not] any grouping of physical microstates within a system may count as identifying digits. Physical microstates have to be grouped in very regimented ways in order to identify a system's digits. Only very specific groupings of physical microstates are legitimate. To a first approximation,[17] physical microstates must be grouped in accordance with the following condition. (p. 127)

Piccinini goes on to describe that condition—essentially that two microstates belong to the same equivalence class if they are treated in the same way.

But as he notes,

> This is not to say that for any two [distinct] input types, a primitive component always generates outputs of different types. On the contrary, it is common for two computationally different inputs to give rise to the same computational output. For instance, in an AND gate, inputs of types '0,0', '0,1', and '1,0' give rise to outputs of type '0'. … [What is important is that when digit tokens representing] different types are supposed to generate different output types, … the differences between [the] digit [tokens] must suffice for the component to differentiate between them, *so as to yield the correct outputs*. [emphasis added] (p. 129).

Note that in the preceding, *digit*, *type*, and *digit type* refer to the abstract type that the equivalence class of microstates represents—thereby stretching *digit* to cover the entire range from low-level physicality to a high-level type abstraction.

[17] As far as I can tell, no more detailed description of how microstates are to be grouped is provided.

The preceding, I believe, are reasonable statements of conditions and properties that must hold for a physical computer to implement abstractly defined computations—which, recall, was the original objective. As the final phrase of the extract says, it is the abstract function that determine whether the implementing device is operating correctly.

5.8 The Bottom Line

Recall that Piccinini posed the problem as follows.

> [U]nder what conditions does a concrete, physical system perform a computation when computation is defined by an abstract mathematical formalism? This may be called the *problem of computational implementation*.

There is no question that it is possible to build physical devices that would qualify under this definition as performing computations. Most people carry them around every day. Further, there seems to be little difficulty in determining whether any particular physical device does so, i.e., whether it is working properly.

Piccinini attempted to do more, to characterize aspects of how *any* such a device must *necessarily* operate. The sticking point in completing this task seems to me to be to identify physical characteristics that are required for the implementation of symbols. Medium-independence, were it possible, might do the job. But I am not convinced that Piccinini has explained how medium-independence and physicality can be married.

More importantly, though, I believe that the task Piccinini set for himself is the wrong task. It may seem mysterious to say so, but it doesn't matter how any particular computing device works physically. All that matters is that it satisfies the requirements specified by the abstract mathematical formalism. More concretely, if a computer (or mobile phone) manufacturer offers a purported computing device for sale, the important question is whether prospective customers would be able to use it for its intended purpose, i.e., to compute? Sections 8 and 9 explore this issue further.

To return to Piccinini's constructions, how does his version of mechanistic computing fare with respect to the other three fundamental abstractions?

Sequences Piccinini spends some time on strings as sequences of digits. Yet there is little discussion of what they are physically, how they come into being, how operations on them are performed, and how their integrity is maintained.

States Piccinini talks about physical devices being in states.

> Typically, an abstract digital computation begins with one initial string (input data plus initial internal state), includes some intermediate strings (intermediate data plus intermediate internal state), and terminates with a final string (output plus final internal state). (p. 126)

Like a digit, a state is an abstraction. I found no discussion of what it means for a physical computing system to be in a state. Even if one grants that physical devices can be in (perhaps quantum) microstates, there is virtually no discussion of

the connection between physical microstates and abstract states and how microstates map to states.

Transitions Virtually all computational abstractions are defined in terms of transitions. For example,

> A Turing machine table can be multiply realized because systems with different physical properties, no matter what functions they perform, can realize the same abstractly specified state transitions. (p. 76)

Like symbols, strings, and states, transitions are abstract. They represent discrete jumps from one state to another. Other than in the quantum realm, such jumps do not occur in nature. I did not see a discussion of how to construct physical devices that can be understood as implementing transitions.

In short, mechanistic computing depends on the same primitives discussed above. Piccinini's mechanistic perspective begins to show us how physical devices might implement these primitives. But I find the explanation incomplete.

Piccinini's work strikes me as a serious attempt to grapple with the fundamental problem: how can physical devices manipulate abstract symbols. That it doesn't work—at least in my view—may suggest that it simply is not possible for a physical device to manipulate abstractions. Perhaps we need another approach. Section 8 offers an alternative. Before going there, Sect. 6 (on Newell's physical symbol systems) and Sect. 7 (on how one might attempt to implement them) clarify what I believe is the fundamental problem.

6 Physical Symbol Systems

A much less detailed but quite insightful approach can be found in Newell and Simon (1976) and Newell (1980). These papers define what Newell and Simon refer to as physical symbol systems: physical systems that are "capable of having and manipulating symbols." According to Newell, who became the primary spokesperson, such systems operate on abstract symbols. They have both the means to organize symbols into expressions and processes that transform expressions into other expressions. (These are essentially the same symbol transformation primitives already discussed—most directly in the discussion of lambda calculus.) Newell (1980) claims that such systems are "the most fundamental contribution so far of artificial intelligence and computer science to … cognitive science." (p. 135)

Newell (1980) is clear that physical symbol systems are indeed physical. [Emphasis in the original.]

> I have been careful always to refer to a *physical* symbol system. (p. 141)

But he is also clear that the symbols on which they operate are abstract. [Again, emphasis in the original.]

> [A physical symbol system memory] is composed of a set of symbol structures,[18] {E_1, E_2, … E_m} [composed of] a set of *abstract* symbols, {S_1, S_2, … S_n}. … The same symbol, e.g., S_k, can occupy more than one [position] in a structure and can occur in more than one structure. (p. 142)

To his credit, Newell offers a solution to how "the same symbol" can occupy more than one position in an expression and can occur in more than one expression.

His solution involves what he calls a "generalized"[19] physical symbol system. (p. 168) In such a system:

- Expressions are structures containing symbol *tokens*.
- Symbols are *abstract types* that express the identity of multiple tokens.

As discussed above, it is a misuse of terms to say that an abstract expression contains symbol *tokens*. The term we have adopted is symbol *occurrences*. I doubt that Newell would object to this alternative.

6.1 Newell's Symbols are Designators

For Newell, a symbol's primary function is to refer to a value, which is distinct from the symbol. He requires that the following relations and operations be defined for symbols. (p. 168)

- *Designation*: a relation between a symbol (type) and the value it symbolizes. [Newell considers this fundamental. It is what "gives symbols their symbolic character, i.e., which lets them stand for some entity." (p. 156).] The designation relation occurs within a computer. It is a simple example of symbol grounding.[20] Unlike the more general symbol grounding problem, designation within a self-contained symbol system generally poses few problems. Section 7 discusses this in more detail.
- *Assignment*: an operation that gives a symbol (type) something to designate.
- *Interpretation*: the retrieval of the value a symbol (type) designates. Newell later makes this concrete as an *access* operation.

> Newell considers the preceding to be fundamental to the notion of a symbol.
>
> It is difficult to envision a notion of symbol that does not embody some version of these capabilities, and perhaps much more besides. (p. 168)

[18] Newell uses the term *symbol structure* as synonymous with *expression*.

[19] Generalized in that it represents any physical symbol system, not just the particular example discussed earlier in the paper.

[20] See, for example, Bringsjord (2015), Cubek et al. (2015), Günther et al. (2018), Müller (2015), and Taddeo and Floridi (2007), to which Bringsjord (2015) is a response.

6.2 Newell's Symbols Differ from Those in the Lambda Calculus and Turing Machines

In the lambda calculus, and in beta-reduction in particular, a symbol is (a) assigned a value after which (b) that value replaces each occurrence of the symbol in an expression. Thus interpretation occurs immediately after assignment; obviating the need for an ongoing designation relationship. As discussed in Sect. 7, in most programming languages, including those based on the lambda calculus, rather than eliminating designation, assignment creates a designation relationship, which is queried whenever a symbol's interpretation is required.

Symbols in Turing machines typically function as opaque markers: they are like letters of an arbitrary alphabet. To use Newell's term, they do not designate.

In both the lambda calculus and Turing machines one of the fundamental operations is to match up symbol occurrences. In the lambda calculus, one must match a symbol occurrence whose symbol is given a value during the first step of beta-reduction with occurrences of that symbol in the expression in which the assigned value must replace those occurrences. In Turing machines, when determining which rule applies, one must match up the occurrence of a symbol in a rule with the occurrence of a symbol under the Turing machine's read/write head.

Church and Turing simply assume that such a matching can be performed and do not discuss how it is carried out. For our purposes, the question of how one performs those matching operations turns out to be fundamental: given two symbol occurrences, how does one determine whether they are occurrences of the same symbol (type)? Section 7.2 examines that question.

6.3 Newell's Symbols are Like Identifiers in Programming Languages

Newell's symbols, really symbol occurrences, are virtually the same as identifiers in Lisp, Newell's preferred programming language. In fact, one can read much of Newell's discussion of physical symbol systems as a somewhat abstract description of—and paean to—Lisp.

> The type of system we have just defined is not unfamiliar to computer scientists. It bears a strong family resemblance to all general purpose computers. If a symbol manipulation language, such as LISP, is taken as defining a machine, then the kinship becomes truly brotherly. (Newell and Simon, p. 116)

Lisp itself is a formulation of lambda calculus in a way that makes it useful for practical programming. As we saw above, Church simply assumes the notion of symbol when describing the lambda calculus. But like all programming languages, Lisp is implemented on physical machinery. So anything a programming language that runs on a physical computer does must be something that can be done physically. Such an operational programming language is then a physical symbol system—Newell's point. Here are two issues this raises—which Newell didn't address.

- How does a physical symbol system represent a symbol type to itself?
- How does a physical symbol system determine for a given symbol occurrence for which symbol type it is an occurrence? That is, how does it link occurrence to type?

The following section addresses these questions.

7 Implementing Symbols

To explore the questions just raised, this section examines two issues: symbol types (as entities that designate values) and programming language identifiers (as symbol occurrences). It examines how such features can be implemented and finds that implementation of these features for a given programming language depends on the existence of another programming language. We find that what looks initially like an infinite regress ends at the bit level.

7.1 What Happens When a Program Runs Depends on the Model of the Programming Language in Which it is Written

For Newell, a fundamental role of a symbol is to designate a value. As indicated above, this is what "gives symbols their symbolic character, i.e., [that they can] stand for some entity." Indeed, this is the primary use of symbols in computing.

When one writes,[21]

```
x = x + 1
```

the `x`'s are programming language identifiers, i.e., occurrences of the `x`-symbol type.

Recall that since symbols are abstract, they may have names even though they can't be exhibited or displayed. We generally let a symbol's name serve as its identifier. Thus, each symbol has a single name, i.e., one identifier. So all occurrences of a given symbol appear identical.

The meaning that one attributes to a line of code typically involves a model, formal or informal, explicit or implicit, of what goes on in the computer when the program runs. Such a model allows one to construct meaning in terms of concepts that are more or less at the same level as the language itself. For the example above, one imagines a model in which (a) the `x`-symbol is associated with some value and (b) execution of that line results in incrementing that value.

In short, what happens when a program runs on a computer requires that an operational model of the programming language in which the program is written exists in the computer on which the program is running. The next section discusses, very briefly, how such models come to exist.

[21] In most programming languages = represents the act of assigning a value rather than a predicate asserting equality. In this case the value associated with the `x`-symbol is incremented by `1`.

7.2 How are Programming Language Models Built?

There is no programming language for which programs written in that language run directly on a modern computer, i.e., on its bare hardware. For any program to run on a computer, a prerequisite is that there already be a program running on that computer that implements the model of the programming language in which the program is written.

With modern programming languages, the situation is a bit more complex. Rather than develop bespoke programs that implement the models for each programming languages, there are now frameworks that make it easier to implement those models. The two most widely used frameworks are the.NET Common Language Runtime (CLR) and the Java Virtual Machine (JVM). Each provides a software implemented virtual machine that both (a) hides the details of the underlying hardware, which may differ from one computer manufacturer to another, and (b) facilitates the development of models for a range of programming languages. Programs written in higher-level languages are translated to run on one of these virtual machines, which themselves run on a range of physical computers.

The details of how this all works are not relevant to this paper. What is relevant is that, as indicated above, pre-existing software is required before a program written in a modern programming language can run on modern computers.

7.3 Symbol Tables Enable Symbol Types to Designate Values

Our focus here is on one aspect of programming language models. In particular, we are concerned with the mechanisms that implement the relationships among: symbol types, occurrences of those types, and values designated by those types. These mechanisms are all implemented in software, i.e., in the pre-existing programs upon which all programming languages depend. That these pre-existing programs are themselves written in programming languages will become an important issue. In particular, are other pre-existing programming languages required for them to run? We will get to this question below. For now, let's just accept that the mechanisms that implement the required symbol/occurrence/value relationships are themselves expressed as software. The rest of this section discusses a simplified version of one such mechanism.

One of the most straightforward strategies for implementing the relationship between a symbol and the value it designates uses computer memory as an intermediary. This approach takes advantage of the fact that computer memory is what's known as addressable. When computer memory is built—as hardware—every memory location gets a number, i.e., an "address," by which it can be accessed. One can think of computer memory as something like a Turing machine tape in which the tape squares are numbered sequentially. (This is, in fact, what Turing does. Recall his G function, which maps the ordinal position—r—of a tape square to the symbol G(r) on that square.)

Computer hardware enables one to use memory addresses to deposit information into and retrieve information from these tape-square-equivalent memory locations. It is as if one had a Turing machine without a read/write head but with a tape whose tape squares were accessible by sequence number.

Given memory of this sort, a symbol (type) may be made to designate a value as follows.

- Associate each symbol (type) with a memory location by making that type a synonym for that memory location's address.
- Associate a symbol type with a value, by putting the value into the memory location with which the symbol is associated. (This is sometimes known as the shoebox model of memory.)
- To make a symbol a synonym for a shoebox address, build a table with two columns. The left column contains symbol types; the right column contains the associated shoebox address. Such a table is typically (and unsurprisingly) called a symbol table.

Since symbol types cannot be displayed or exhibited, what is actually in the left column? Just as we use the symbol name, i.e., its identifier, for its occurrences, we can use the same identifier to represent the symbol in the left column of the table. So now we have a table with identifiers, i.e., symbol names, in the left column and memory addresses in the right. Such a table can be understood as associating a symbol with a memory location.

Once such a table exists, we can use it to link symbol occurrences in programs with symbol values. On encountering a symbol occurrence, as in our example above, look in the symbol table for a matching occurrence, i.e., a matching identifier. The memory address in the right column corresponding to that identifier is the address for the symbol's memory location. That memory location holds the value the symbol designates.

Thus in our example above, we use the table to look up the address associated with the symbol whose name is `x`. We then look in the memory location with that address to find the symbol's value. We increment that value and return the incremented value to the same memory location. The preceding is an over-simplification—and grossly inefficient—but it will serve our purposes.

To perform these steps, one writes software that implements a symbol table and that performs the lookup operations just described. In other words, one implements the types and occurrences of one language, say Lisp, via software written in a preexisting language. This has been standard in programming language implementation for three quarters of a century: use one programming language to implement another.

7.4 Implementing Identifiers: Software is Built on a Foundation of *Bits*[22]

So far, so good. But we have not solved the problem of how a physical device can manipulate symbols. The problem is that the implementing language needs symbol types and identifiers of its own. Why? The implementing language builds the

[22] Bits make a strange foundation. Although as abstractions they are less substantial than sand, they are far more enduring.

symbol table that maps the symbol occurrences and types of the higher-level language to memory addresses. The implementing language uses identifiers for the occurrences of the symbol types of the implemented language.

How does the implementing language construct and manipulate those identifiers? For example, to determine the value associated with a symbol occurrence (such as the `x`-symbol occurrence in our example line of code), the implementing language examines the symbol table to find a matching `x`-symbol occurrence. But what are these identifiers? Identifiers in the higher-level language are expressions created by the implementing language. Those expressions are composed of symbol occurrences from the implementing language. In other words, to implement symbol types and occurrences of one programming language one needs an implementing language in which symbol types and occurrences already exist.

The preceding may have sounded complex, but in fact, identifiers are almost always sequences of characters. So in plainer language, the implementing language uses sequences of character occurrences to represent the identifiers of the higher-level language: each higher-level language identifier has a spelling in the implementing language. The spelling consists of character occurrences in the implementing language. Two identifier instances represent the same symbol if they are spelled the same way. In other words, we converted the test for whether two identifiers in the higher level language represent the same symbol into a question of spelling in the lower-level language.

In doing this have we trivialized the problem? No; we haven't solved the problem at all! All we have done is to move the problem from the higher level language to the implementing language. We still don't know where the symbols types and occurrences of the lower-level language come from or how they are implemented.

7.5 Is it Software All the Way Down?

Can one pull the same trick with the implementing language and say that its types and occurrences are implemented in yet some other language? Can one say that it's programming languages all the way down? Unfortunately, no. That wouldn't solve our problem. We can't point to an infinite regress of implementing languages and say "problem solved!" At some point, the symbolic world must be attached to the physical world. There is no infinite regress. So we have yet to answer the question of how to bridge the gap between a physical computer and the manipulation of abstract symbols.

But this apparent trivialization raises questions. How do we know—actually, how does the implementing language decide, whether two identifiers are spelled the same way? Let's look at that more carefully.

What does it mean to look at the spellings of two identifiers to see if they match? One looks at the corresponding character occurrences in the two identifiers and ask for each pair whether they are occurrences of the same character in the implementing language. If all the corresponding pairs of

character occurrences match, we say that the identifiers are occurrences of the same symbol. Otherwise we say that the identifiers and hence the occurrences, are different.

Simple—and trivial; but it raises two issues.

What do we mean by a sequence of character occurrences? As we said earlier, sequentiality is generally taken as primitive. It's not a difficult concept, but it's important to acknowledge that this (or addressability) is one of the foundational abstractions built into computers. So, how is sequentiality implemented?

The easy answer is that sequentiality is implemented by considering sequential memory addresses. So a sequence consists of memory addresses, such as $\texttt{address}_1$, $\texttt{address}_2$, etc., where each of the addresses is numerically equal to the previous address plus 1. (Notice that this assumes the ability to perform this sort of simple arithmetic. Again, this is not difficult, but it is part of the framework we must assume is available.)

Once we know what sequentiality means, how do we decide whether the n^{th} character occurrence in one sequence is an occurrence of the same character type as the n^{th} character occurrences in the other? Isn't that essentially what we started out asking?

Yes, it is. To compare character occurrences we again look at their spellings. What does that mean? Character occurrences are typically represented as bit sequences. So for each pair of character occurrences we look at the sequences of bit occurrences for those character occurrences and see if the bit occurrences match. If for all k the k^{th} bit occurrence of one character occurrence is the same as the k^{th} bit of the other character occurrence, the character occurrences are of the same character. If not, they're not.

And how do we determine whether two bit occurrences are the same? The same question once again? Are we just going down the rabbit hole?

> The rabbit-hole went straight on like a tunnel for some way, and then dipped suddenly down, so suddenly that Alice had not a moment to think about stopping herself before she found herself falling down what seemed to be a very deep well. Either the well was very deep, or she fell very slowly, for she had plenty of time as she went down to look about her, and to wonder what was going to happen next. First, she tried to look down and make out what she was coming to, but it was too dark to see anything. … "Well!" thought Alice to herself, "After such a fall as this, I shall think nothing of tumbling down-stairs! How brave they'll all think me at home! … Down, down, down. Would the fall never come to an end, … when suddenly, thump! thump! down she came upon a heap of sticks and dry leaves, and the fall was over.

We, too, are about to hit bottom.

7.6 Less is Different: Determining Whether Two Bit Occurrences are the Same is Different from Determining Whether Two Character Occurrences are the Same

Determining whether two bit occurrences are the same may seem abstractly to be the same as determining whether two character occurrences are the same—or whether two identifiers are the same, but it is quite different.

Bits are understood as atomic abstractions; we can't break *Bits* open to see if *their* components match up. And bit occurrences are not even symbolic. Bit occurrences may be represented as voltage levels, current levels, levels of light intensity, direction of magnetization or polarization, etc.—or as we saw when discussing Piccinini, as a switch being either on or off. Furthermore, two bit occurrences may be considered the occurrences of the same bit value, i.e., both 1's or both 0's, even if their representations are completely different. We have landed thump! thump! at the bottom of the rabbit hole.

How do we determine whether two bit occurrences are the same value. We could, of course, look at the physical implementation of bits, i.e., at voltages, etc., but that raises questions about physics, chemistry, materials engineering, electrical engineering, etc. and would force us to face the same sorts of questions Piccinini took on.

Rather than pursuing a physical analysis of bits, our approach will be to let bits and their occurrences remain abstract. The question, of course, is how we can do that and still explain how physical devices are able to manipulate them. The answer involves what in computer science are known as abstract data types.

8 The Bit as an Abstract Data Type

This section first offers a quick overview of the history and terminology of computational data types. It then defines abstract data types and offers some everyday analogs such as user manuals. Finally, it defines a Bit abstract data type and discusses the division of labor between programmers and computer engineers.

8.1 From Hardware Structures to Collections of Operations

Programming languages organize the objects they manipulate into categories known as data types. The original data types corresponded to categories that were defined by computer hardware. In particular, early computer hardware provided distinct ways to represent (a) integers, (b) floating point numbers (which are ways to represent real number approximations—recall the earlier discussion of computational science), and (c) characters. *Integer*, *Floating Point*, and *Character* became the early software data types.

- Integers—within a fixed range—were represented as sequences of (frequently) 32 bits—or 16 bits in earlier computers. These were considered to be either a

sign bit along with 31 bits of magnitude or a 32 bit representation of a two's complement binary number.
- Floating point numbers were represented by a sign bit, a fixed-size (in terms of the number of bits used) mantissa for the number's normalized value, and a fixed-size exponent.
- Characters were represented as 8-bit sequence of bits, which came to be known as bytes.

For each of these categories of data, computers included instructions to manipulate data in these categories. For example, there were instructions for performing arithmetic operations on integers and other instructions—because the data representations were different—for performing *the same* arithmetic operations on floating point numbers. See Chua (2018), for example, for a tutorial on these data type representations and operations.

Data types of this sort were not considered abstract and were not intended as a computer science analog to linguistic or philosophical types. On the contrary, the concreteness of the representations was generally considered the essence of the data types.

It is not quite that simple, though. I refer to the representations as concrete, and people in the field considered them concrete. But notice that in all three cases, the representations consisted of arrangements of bits. And bits are abstract. So it would be more accurate to say that these data types were represented as fixed arrangements of bits. Of course the notion of an arrangement is itself an abstraction: the primary arrangement being a sequence. So concrete as these early data types seemed, they were built on abstractions.

These data types did not appear out of thin air. They were based on pre-existing, but generally unformalized, notions about integers, real numbers, and characters. It was taken for granted that everyone understood well enough what were meant by integers, real numbers, and characters so that no more needed be said about them.

It was also generally taken for granted that the representations were close enough approximations of the abstractions that one could often ignore the differences. Yes, computer-based integers were a finite set even though actual integers are infinite. But computers themselves suffer the same sort of limitations. A computer is in reality a finite automaton even though we treat it as if it were a Turing machine. Consequently, once the computer hardware mechanisms were developed, the original intuitions about integers and real number approximations were often set aside, and software was understood as operating on these hardware-defined data types.

This often came back to bite software developers. For example, if `X` is a very large real number (approximated as a floating point number), it was often the case that the real number `X + 1` was represented identically (in terms of bit values) as `X`. This was the case because the precision of the floating point mantissa was inadequate to represent the difference between the two: only the high-order digits could be represented. As one might imagine, this produced difficult-to-find software bugs.

As programming languages and programming language theory grew more sophisticated, so did the computer science notion of a data type. In an important step, Wirth (1971) introduced "programmer-defined data types" in the programming

language Pascal. This feature enabled software developers to define a data type as a collection of constant values: a *traffic-light-color* type might be defined as the color names *red, yellow,* and *green*.

As far as the programming language was concerned these were symbols (from an arbitrary alphabet) and their occurrences. Although they were represented as integers (selected by the programming language), the software developer was unaware of the specific integers associated with each name and was free to manipulate the names as opaque, i.e., non-designating, symbols. Defining a data type of this sort did not imply a semantics for the symbols. Their semantics was an implicit consequence of how the software treated the symbols.

8.2 Abstract Data Types

A breakthrough occurred in 1974 when Liskov and Zilles (1974) defined what they called an *abstract data type* (ADT).

> An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects. (p. 51)

The essential difference between an abstract data type and earlier types was that an abstract data type ignores both representations and concrete values. The essence of an ADT consists of the operations that can be performed on elements of the type.

In defining an ADT in this way Liskov and Zilles inverted the relative importance of representation and operation. In a traditional data type, as described in the previous subsection, the structure that the computer provides for representing members of a given type provides the primary driver: integers were represented in one way, floating point numbers in another, and characters in a third. Even programmer-defined types were understood to be represented by integers, although the particular integer associated with each symbol was not relevant. In contrast, an abstract data type is defined by a collection of operations, not by a representation structure. This inversion of focus was the primary ADT insight.

In their original form abstract data types were often specified formally. The standard example involves an abstract specification of a stack—a last-in-first-out data structure. The standard stack operations are `push` and `pop`. The specification includes the following constraints. (Although they can be expressed in a formal language, they are expressed here informally.)

1. The object one `pop`s off a stack is the object that was most recently `push`ed onto the stack.
2. The stack that remains after a `pop` operation is identical to the stack as it was prior to the most recent previous `push`.

An object that offers `push` and `pop` operations satisfying these constraints may be considered a stack.

Elegant as this approach is, most software components are too complex for a complete specification, formal or informal. Instead, component specifications are now

given in terms of what are known as Application Programmer Interfaces (APIs). An API is a list of available operations along with, for each, an intuitive, description of the expected behavior. This generally turns out to be adequate.

8.3 Generalizing the ADT Beyond Software

ADTs were intended for software data types, but the ADT definition does not mention software. The definition can, and should, be understood more generally. Consider the relationship between a user manual and the system it describes. A user manual is like an ADT: it describes the functionality of the system for which it is a manual in terms of the operations a user can perform on that system.

If a user manual describes, say, a kitchen appliance, the user manual may be considered something like a type, and each physical appliance may be considered something like a token of that ADT. Similarly, if a user manual describes, say, a software system, e.g., an "app," each copy of the app may be considered an instance[23] of the ADT. The ADT, in turn may be understood as a functional description of these physical devices or apps.

As just described, it is not clear which comes first. Is the user manual written in advance and the device or app built to behave as the user manual says? Or is the device or app built, followed by the writing of a user manual that describes how it behaves? Either is possible. The relationship between a software component and its abstract data type and/or user manual may be quite fluid, with either side of the relationship influencing the other.

When developing a product (either software or a physical device), it is common practice to write something similar to a user manual before embarking on the actual construction of the product. A functional specification (alternatively a requirements specification) describes the intended behavior of an item that is yet to be produced (Turner 2011). "Specifications are taken to be prescribed in advance of the artifact construction; they guide the implementer." (Turner and Angius, 2017). As Turner and Angius also point out, what we just described is an idealization.

> A specification is not fixed throughout the design and construction process. It may have to be changed because a client changes her mind about the require-

[23] As programming languages grew more sophisticated they added a feature similar to abstract data type. It is known as a *class*. A *class* is similar to what philosophers might refer to as an ontological *kind*, i.e., a category of things that are naturally grouped together, such as dogs, schools, etc. Programmers define the classes they need for a particular program. A University scheduling program, to take a well-worn example, may include *class*es for students, instructions, courses, sections, lecture rooms, grades, etc. When a programmer defines a *class* she characterizes how it behaves—similar to how one defines an abstract data type. In the case of programming language *class*es, the behaviors are specified by what are known as methods. These method specifications are written in software, not in a formal language such as predicate calculus. A method is defined by the program that is executed when the method is activated.

An important feature of *class*es is that software may create instances of them, e.g., a particular course, or a particular student, etc. The program may then manipulate these instances according to the operations the *class* defines: e.g., one may enroll a student in a course. As I just did, the term *instance* is typically used for these cases. I will often use that term to mean a particular case of a defined category.

ments. Furthermore, it may turn out for a variety of reasons that the artifact is impossible to build. The underlying physical laws may prohibit matters. There may also be cost limitations that prevent construction. Indeed, the underlying definition may be logically absurd.

Such a functional specification serves as an ADT for the item to be produced. Crucially, a functional specification does not discuss the design, either physical or abstract, of the item to be produced. Its focus is on the behavior of the intended system and how a user may interact with it. Like an ADT, a functional specification defines a category of items in terms of the operations they are able to perform —or at least the external manifestations of those operations. In short, ADTs, user manuals, and functional specifications may all be understood as ways to characterize a category of items in terms of operations they may be required to perform.[24] More informally, ADTs, user manuals, and functional specifications characterize how an item may be used.[25]

8.4 A *Bit* ADT

Our original motivation for discussing abstract data types was to define the *Bit* as an abstract data type. This section returns to that task. Let's consider what a *Bit* ADT might look like.

1. *Values:* Every *Bit* instance is one of two given values. These values are themselves abstract. Like other abstractions, they cannot be displayed or exhibited. But, like other abstractions we can name them. How we name them doesn't matter—1 or 0, on or off, high or low, True or False, etc. The names are not the values; they are means we provide ourselves to refer to the abstract values.
2. The following operations must be defined.

    (a) to determine which of the two possible values a given *Bit* instance is;
    (b) to determine whether two *Bit* instances are the same value;

[24] When a user employs an operation listed in the API of a software element, the software element behaves in some way. Since the behavior is symbolic it would be misleading to call it physical. On the other hand, since the behavior involves the execution of portions of the software entity by a physical computer, the behavior is fundamentally physical. This presents a terminological difficulty. Something physical occurs, but the result is understood symbolically. I am using the term *discernable* for this sort of situation. A crucial element of the behavior is that the user of the API operation is able to discern that it occurred.

Discernibility does not imply that the effect of the behavior is immediately apparent. For example, entering something into a database is not immediately discernable. It is discernable in that a query to that database reveals that the information is in it. Similarly, the effect of `push`ing something onto a stack is not immediately discernable. It becomes discernable when a `pop` operation pops that element off the stack.

[25] If taken too rigidly, such a perspective risks seriously mischaracterizing an item. What one can do to an item and how it will respond is generally well defined. How an agent can make use of an item depends on the imagination and creativity of the agent—as is revealed by the ever-popular challenge of listing 100 things one can do with some item, which may often include using it as a doorstop or paperweight.

(c) to create a *Bit* instance with a given value;
(d) to build larger structures, e.g., sequences, from individual *Bit* instances;
(e) to perform the traditional logical operations on *Bit* instances and thereby produce other *Bit* instances.

We can now answer our question about how we determine whether two bit instances are the same bit value: make use of the operation described at (2) (b). This tells us that there is an operation, often notated as ==, which will tell us whether two bit instances, e.g., $\texttt{bit}_1$ and $\texttt{bit}_2$ are the same value.

$\texttt{bit}_1 == \texttt{bit}_2$

When we perform this operation we will get a result bit instance, which will be either `True` or `False`, which tells us whether $\texttt{bit}_1$ and $\texttt{bit}_2$ are the same bit value.

Did the preceding answer the question: how does one determine whether two bits are the same value? All we said was that the *Bit* ADT guarantees that *there is* an operation (==) that answers the question. The ADT doesn't say how that operation works; it doesn't say where the responsibility lies for performing that operation or for making sure it produces the right answer. It just tells us that there is such an operation and that it answers the question we want answered.

That's exactly what an ADT does. It specifies operations that must be provided whenever an ADT is realized. It doesn't say how the operations work. It only says that whenever an object claims to implement an ADT, it must provide the specified operations. That's also what a user manual or functional specification does. User manuals and functional specifications don't say how the behaviors they describe are performed. They only specify how an item must behave if it is to adhere to the description provided by a user manual or functional specification.

The *Bit* ADT is one small but fundamental portion of the user manual for all computers. For a physical device to be considered a computer it must implement the *Bit* ADT. So the answer to the question of what implements the *Bit* ADT is that it is done by the computer itself. The computer hardware implements the *Bit* ADT. Software developers need not be concerned about how the hardware works or what techniques it uses. Software developers need know only that every computer makes available Bits that satisfy the description of the *Bit* ADT. With the assurance that those operations will be performed as specified, software developers can write software that makes use of *Bit* operations. Once software developers can use the defined *Bit* ADT operations to manipulate *Bit* instances—and once they also have the other abstractions such as sequentiality, state, and state transition—they can then build up the enormous world of software we now have.

8.5 How Engineers Make *Bits*: Converting Symbols to Matter

You may be thinking, that's all well and good, but how *is* the *Bit* ADT implemented in a physical computer? We are still talking about abstractions; we have yet to explain how physical machinery is capable of performing operations on abstractions. Saying that computer hardware implements the *Bit* ADT doesn't seem to add any insight.

But it does add insight. Church and Turing each gave us an abstract foundation for general purpose computing. The most fundamental element for each is the *Symbol*, and the simplest *Symbol* is the *Bit* with its two values. The question these giants left open is how a physical device can bring abstract *Symbol*s into existence and interact with them—or what that even means.

Piccinini described features that physical computing device must implement. These included symbols in the form of what he called digits. In the end he did not explain how digits are implemented, only that they must obey the rules defined by abstract computational formalisms, i.e., that they must implement abstractions such as a Bit ADT—although he didn't put it in those terms.

Newell and Simon identified the issue clearly: how does a *physical* system manipulate *Symbol*s? Strangely, at least to my mind, Newell and Simon did not remark on how extraordinary such a device would be. A physical symbol system would bridge the gap (chasm?) between the physical world of hardware and the symbolic world of software. As we've been asking all along, is it even possible to bridge that gap?

As I've said, my answer is that it is possible: engineers build computer hardware to do exactly that. The reason this may seem like a non-answer is that by framing the question as we have, we have confused the issue. It is not the case that computer hardware converts matter to symbols. That would not make sense. Computers cannot create abstractions. What really happens is that computer hardware converts symbols to matter, a much easier task.

To see what this means, consider a simple light switch (See Fig. 1). Suppose one asks how a light switch converts physical materials, i.e., plastic, glass, wires, electricity, etc., into the symbolic conditions of a light being `on` or `off`. That is essentially the question that Piccinini attempted to answer: how does a physical computer produce symbolic computation? And that's what seems so special about physical symbol systems. But that's the wrong question. A light switch and the associated materials doesn't create `on` or `off` abstractions. What could that possibly mean?

The correct question is: given that the human brain already functions in terms of abstractions—such as `on` and `off`—how might engineers build physical devices so that we may associate the abstractions we already use with phenomena in the physical world?

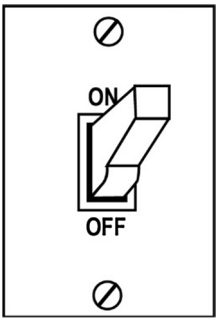


Fig. 1 A wall toggle switch Image from clipartxtras.com (https://clipartxtras.com/): https://img.clipartxtras.com/2c02c44d34c4f9fe5dab7a982b977acc_tales-of-a-garlic-and-onion-lover-mix-up-monday-light-switches-light-switch-drawing-easy_371-544.jpeg

As human beings, we are able to characterize the abstract functions we care about. For any particular abstraction—such as a light being on or off—we can develop an abstract data type, either explicitly or implicitly, that describes how that abstraction functions. We then build a physical (or software) artifact that behaves as the ADT requires. A wall switch, for example, doesn't "turn a light on or off." A wall switch is a component of a circuit that under certain conditions allows an electric current to flow through a device, like a lightbulb, capable of emitting photons. The photon-emitting device is constructed in such a way that a flow of electricity through it results in the emission of energized photons, which we see as a light being on. The switch, when flipped, opens or closes the circuit through which electricity to the photon-emitting device flows.

So the switch doesn't "turn a light on or off." It allows a user to change the state of the world so that photons either are or are not emitted from a photon-emitting device. The concepts *light-on* and *light-off* are human abstractions. The electric circuit, the photon-emitting device, and the switch are constructed in such a way that we find it natural to associate those concepts with states of the world that we can control by flipping a switch.

Similarly, to create *Bits*, engineers start with the *Bit* ADT and build hardware that behaves as the ADT requires. It's not the case that *Bits* magically spring to life from computer hardware. Computer hardware behaves as a *Bit* ADT requires—much as a wall switch behaves as our understanding of switches require. When understood this way, the job is much more straightforward. The ADT perspective tells us that what matters are the ADT operations, i.e., the services, not the physical devices that implement the services.

9 ADTs from the Perspective of Their Users: Affordances, Thought Externalization, and Symbol Grounding

This section extends the thought that ended the previous section by examining services from the perspective of their users. We begin with the notion of an affordance, which was originally understood as a physical API that enabled agents to make use of some external entity for the agent's own purpose. (The wall switch is a good example.) We then generalize that idea and see affordances as means (a) that are external to an agent and (b) that give the agent access to services that satisfy a functional need. We will also see that in so doing affordances enable agents—at least agents that are capable of forming concepts—to ground symbols.

When we first introduced the ADT we noted that an ADT perspective inverts the relative importance of physical representation and function. What's important—at least to those who make use of ADT instances—is not how physical materials are combined to build new physical entities. (Do you really care what microwaves are or how your microwave oven generates them?) What's important is that instances of simpler ADTs may be combined to create more sophisticated ADTs. (All you really care about is that your microwave oven does what it is supposed to do, i.e., heat food—and does it safely and in a way that you find convenient.) What matters is not the thing but the service.

As a somewhat fanciful example consider ride-hailing services such as those (currently) provided by Uber and Lyft as well as those soon to be provided via autonomous vehicles by companies such as Alphabet's Waymo. Ride-hailing services do not define a new ADT; they implement an existing personal transportation ADT in new ways. Although the new implementation may differ from the old, as far as the user is concerned, the service is the same.

Piccinini struggled to find a necessary common physical core for all implementations of computing. I suspect it would be similarly difficult to find a necessary common physical core for all implementations of personal transportation services. What physical core do the following two personal transportation service implementations have in common: (a) a traditional taxi service and (b) a hypothetical 2050 service that accepts mentally transmitted requests for transportation and then teleports customers to desired destinations. Although the service is essentially the same, the implementations will have very little in common at the physical level. But from a user perspective, they are quite similar. Similarly, there are multiple implementations of the *Bit* ADT. They also have little in common physically.

This section has taken us through a two-step journey (a) from a *thing* perspective (How can physical devices operate on symbols?) (b) to a *service* perspective (How can one implement a symbol manipulation ADT?) and then (c) to a *thought* perspective (Symbol manipulation is an abstraction. Because we understand what we mean by symbol manipulation, we can build services to do it for us).

9.1 Affordances as Concrete APIs

ADTs characterize things in terms of the operations that can be performed on them. This notion has a parallel in psychology. In the same decade in which the Liskov and Zilles paper appeared, Gibson (Gibson 1977, 1979) described what he called affordances: properties of things that are relevant to the ways in which agents can interact with them. In his writings about artifact design, Norman (1988, 2013) introduced the term to a much larger audience. He argued that objects should be designed with affordances that users understand intuitively. For example, most people know at a glance that a car's steering wheel enables the driver to change the forward direction of the car by turning the wheel. Norman may be best known for criticizing poorly designed doors. When approaching a door with a waist-high bar, one doesn't know whether to push the bar or to pull it. A better designed door would present a push-plate if the user is supposed to push the door and a handle for the user to grab if the door is intended to be pulled.[26]

The notion of an affordance can be generalized to encompass a very wide range of interactions. For example, plants depend on insects to transport pollen as part of their fertilization cycle. That insects respond to the form and color of flowers makes insect sense organs affordances for plants that rely on these insect activities. The plant creates flowers with the appropriate shape and odor to trigger the desired

[26] This cute 5-minute video illustrates the problem. https://youtu.be/yY96hTb8WgI.

behavior on the part of the insect. A plant thus interacts with an insect through the insect's (sense organ) affordances.

In a software context, an API lists the affordances that a software entity makes available.

With such examples in mind, one might reasonably argue that *every* non-forceful interaction, i.e., every communication between identifiably distinct entities, takes place through an affordance. An affordance becomes the property of an entity that enables some other entity to interact with it.

Affordances play a central role in the interventionist approach to causality (Pearl 2000; Woodward 2003). According to an informal characterization, X has a causal relationship to Y if wiggling X reliably results in Y wiggling. Or as Woodward (2014) put it, X has a causal relationship to Y means that X can serve as a remote control for Y. Although affordances do not necessarily imply remote control, they enable non-forceful external control.

9.2 Affordances and Symbol Grounding

This section examines affordances as links between an abstraction in the mind of an agent—at least agents that have minds capable of forming concepts—and a discernable action on the part of some other entity.

An affordance enables an agent to realize a functional need through the behavior that some other entity performs when the agent avails itself of one of that entity's affordances.[27]

Notice what we just said. An agent has a functional need. The functional need is an abstraction, whether or not it is conceptualized by the agent with the need. The agent satisfies its functional need by manipulating an affordance offered by an external entity. The agent's manipulation of the affordance triggers some behavior by the external entity that results in the satisfaction of the agent's functional need.

Considered from the agent's perspective an affordance represents an action or activity an agent can trigger by manipulating the affordance. Consider again a wall light switch. A wall light switch is an affordance that an electrical system offers to users. When a user flips the switch, the associated light goes on or off—assuming all the other elements are working properly. From the perspective of the switch flipper, a light being on or off is an abstraction. A light switch enables an agent to change the light from one abstract state to the other. A light switch gives an agent control over something she understands as an abstract property of the physical world.

For its users, an affordance becomes a physical representation of an operation they can trigger.

Most engineered artifacts implement user functional needs. The development of such artifacts occurs in a series of steps.

[27] When a plant uses insect sense organs as affordances, the plant realizes a functional need (pollen transportation) even though that functional need is not reified by the plant as a concept. See Dennett (2017) for an extended discussion of what he refers to as "competence without consciousness".

1. An engineer identifies a functional need and imagines a product or device that could satisfy that need.
2. She writes a functional specification to describe more fully how the product or device works.
3. A physical device or a software element is then constructed to realize the functionality described in the functional specification.

Customers buy the product or device, which enable them to satisfy their functional needs by availing themselves of the affordances offered by the product or device.

To design and build a product or device, engineers generally write a functional specification that characterizes how a user may operate it. Once built, the functional specification has served it purpose and is no longer needed. As far as the user is concerned, the affordances offered by the product of device should[28] signal how to make use of it.

Once built and in the hands of a customer, a product or device lets users link their functional needs—whether explicitly conceptualized or not—to the affordances offered by the product or device.

As an affordance, a light switch enables an agent to change the physical world to conform to an abstraction. Affordances are thus *symbol grounding mechanisms*. They enable an agent to connect an idea to something in the physical world. Well-designed affordances should eliminate the need for user manuals and allow users to turn ideas directly into actions—and hence to changes in the physical world.

10 Final Thoughts

When we originally raised the question of converting physicality to abstraction, it may have seemed like we were asking for magic. It's unreasonable to ask a physical device to create and manipulate abstract symbols. But now we see that like most magic, the magic of physical symbol systems is surprisingly mundane. What's really happening is that instead of conjuring up bits, we are implementing abstractions.

Everything in the world of computation can be built of (abstract) bits. That is, if we had bits the rest would follow. But once we know what we need as a starting point, i.e., something that behaves as described by a *Bit* ADT, the rest is standard engineering: build hardware that implements the *Bit* ADT. Computer hardware is the physical realization of the (abstract) idea of a *Bit*.

In what he referred to as his farewell film Hayao Miyazaki (2013) popularized the saying "Engineers turn dreams into reality" by putting it in the animated mouth of Italian aircraft designer Giovanni Battista Caproni. If one can imagine something, it may be possible to find a way to realize the idea physically. "Engineers turn dreams

[28] According to Norman, the affordances offered by a product or device will signal how to use it if they are well designed. Most commercial products come with user manuals. Most users don't read user manuals; they expect the produce's affordances to provide sufficient guidance about how to use the product.

into reality"[29] is a poetic way of saying that engineers (and software developers) externalize their ideas as systems. (See Abbott (2008, 2009, 2018, and 2019) for additional discussion about externalizing ideas.)

In fact, it's a bit more sophisticated than that. Engineers build systems that allow *users* to externalize their own ideas, i.e., to turn their own dreams about what they want to happen into the reality of having them happen.

- Want a cup of coffee? Put coffee beans and water into a coffee maker and press the *Start* button. The coffee maker brews coffee. But it doesn't really. It performs physical operations—grind the beans, heat the water, etc.[30]—that result in something we understand as coffee.
- Similarly for clean clothes. Put dirty clothes into a washing machine, adjust the settings, and press the *Start* button. The washing machine cleans the clothes. But it doesn't really. It performs a number of physical operations that result in what we understand to be clean clothes.

Engineers build devices that allow users to turn their own dreams—of fresh hot coffee or of brightly colored clean clothes—into reality.

Appendix: Unicode. This Appendix is Brought to You by the Letter 'A'

This section examines how character types are treated in software.

For software, text, and data to be portable from one computer to another requires agreement on how letters, digits, etc. are to be encoded. (Even if each file of software, text, or data were always accompanied by a translation table that specifies how that file refers to letters and digits, the table itself must refer to the various letters in order to specify their file-specific representations.) Because portability is so important, there is now agreement, virtually worldwide, on what is known as the Unicode Standard. The Unicode standard provides a way to refer to any character in any of the world's written languages. Here is how it describes itself (Unicode 2017).

> The Unicode Standard is the universal character encoding standard for written characters and text. It defines a consistent way of encoding multilingual text that enables the exchange of text data internationally and creates the foundation for global software. …
>
> The Unicode Standard specifies a numeric value (code point) and a name for each of its characters. …

[29] That engineers turn dreams into reality is not original with either Miyazaki or Caproni. But I have been unable to find an original use. Turning dreams into reality turns out to be a fairly common theme. For example, a 1974 ad for Sony sound systems (Sony 1974) lets readers know that the product exists because, as the ad says, "engineers dream, and at Sony they turn their dreams into reality.".

[30] These are also abstractions. The coffee maker doesn't "grind beans;" it sets cutting blades into motion, which results in beans being ground. It doesn't "heat water;" it causes electricity to flow though a device that when activated produces heat. Etc.

> The Unicode Standard contains 1,114,112 code points [the (hexadecimal) digits 0 to 10 $FFFF_{16}$] most of which are available for encoding of characters. … The overall capacity for more than 1 million characters is more than sufficient for all known character encoding requirements, including full coverage of all minority and historic scripts of the world.

The Unicode Standard even includes emojis.[31] This might seem difficult because an emoji is a small image, and emoji images are not standardized. According to the Standard,

> The mark made on screen or paper, called a glyph, is a visual representation of [a] character. The Unicode Standard does not define glyph images.

In other words, the software that displays text is free to choose an image for each character. This makes sense in that each font face prescribes its own visual images for characters.

The same holds for emojis. For example, 🧬 is the recommended Unicode glyph for the DNA emoji—although, other glyphs are possible.

What does the Unicode Standard tell us about the letter 'A'?

> The character identified by a Unicode code point is *an abstract entity* [emphasis added], such as "latin capital letter a" or "bengali digit five."

Note that "latin capital letter a" is considered an abstract entity.

The Unicode Standard does not include definitions for the characters it catalogues. It does not include anything like "the letter 'A' is the first character of the latin alphabet." Each Unicode character is no more or less than an assigned name (e.g., "latin capital letter a") along with a Unicode code point, i.e., a number expressed in hexadecimal notation. The capital latin letter a, for example, has this entry.

latin capital letter a (U+0041)

When a Unicode encoded character appears in a text, all that appears is the character's code point value. The software that displays it selects the glyph to display. In other words, when working on a document in a word processor, the content of the document within the computer is a sequence of code points. It is up to the word processing system to decide the glyphs to use to display those numbers.

A corollary is that characters themselves, i.e. characters as abstractions, cannot be displayed or exhibited. Abstractions have no visual representations.

Given this framework Wetzel's example, ∃x(Ax & Bx), would be expressed as follows in Unicode. (The latin characters are shown above the code point hex values.)

∃	x	(	A	x	<space>	&	<space>	B	x	)
U+2203	U+0078	U+0028	U+0041	U+0078	U+0020	U+0026	U+0020	U+0042	U+0078	U+0029

[31] See http://unicode.org/emoji/.

The three occurrences of x are represented by the Unicode code-point U+0078. In other words, the symbol x itself does not appear in the expression. What appears is the identifying number of the symbol x.

In other words, ∃x(Ax & Bx) is not (only) a statement in predicate calculus; it is also a sequence of characters—which can be interpreted as a predicate calculus statement. In particular, the Unicode code point U+0078 refers to the character x, which refers to an abstract predicate calculus symbol, which occurs in three places when the sequence is taken as a predicate calculus statement.

As before, we are implementing one language by using an existing language. In this case we are implementing the language of predicate calculus by using a language that includes the Unicode Standard for encoding characters.

We can ask the same sort of question we asked earlier. How does one determine that the three occurrences of U+0078 are, in fact, the same Unicode code point? Since code-point numbers are represented as hexadecimal characters, we are again back to where we started, this time asking how one compares two hexadecimal characters? The answer is the same as before. To determine whether two hexadecimal characters are the same, one compares their bit representations. Of course the same question arises again at yet lower level: how does one tell whether two bits are the same bit value? As before, as discussed in Sect. 8, we depend on a hardware-implemented *Bit* ADT.

References

Abbott, R. (2008). If a tree casts a shadow is it telling the time? *International Journal of Unconventional Computing, 4*(3), 195–222.

Abbott, R. (2009). Bits don't have error bars: Upward conceptualization and downward approximation. In I. Poel & D. Goldberg (Eds.), *Philosophy and engineering: An emerging agenda* (pp. 285–294). Berlin: Springer.

Abbott, R. (2018). Meaning, autonomy, symbolic causality, and free will. *Review of General Psychology, 22,* 85–94. https://doi.org/10.1037/gpr0000125.

Abbott, R. (2019). A software inspired constructive view of nature. *Presented at the 2016 conference of the international association for computing and philosophy*. To appear in selected papers from that conference. Springer.

Blum, L., Shub, M., & Smale, S. (1989). On a theory of computation and complexity over the real numbers: *NP*-completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society, 21*(1), 1–46.

Bringsjord, S. (2015). The symbol grounding problem… remains unsolved. *Journal of Experimental & Theoretical Artificial Intelligence, 27*(1), 63–72.

Chua, H.-C. (2018). Teaching notes, National Taiwan University. https://www3.ntu.edu.sg/home/ehchua/programming/java/datarepresentation.html.

Church, A. (1932). A set of postulates for the foundation of logic. *Annals of Mathematics, 32*(33), 346–366.

Cubek, R., Ertel, W., & Palm, G. (2015). A critical review on the symbol grounding problem as an issue of autonomous agents. In *Joint German/Austrian conference on artificial intelligence (Künstliche Intelligenz)* (pp. 256–263). Cham: Springer.

Cuffaro, M. E., & Fletcher, S. C. (Eds.). (2018). *Physical perspectives on computation, computational perspectives on physics*. Cambridge: Cambridge University Press.

Dennett, D. C. (2017). *From bacteria to Bach and back: The evolution of minds*. New York City: WW Norton & Company.

Gibson, J. J. (1977). The theory of affordances. In R. E. Shaw & J. Bransford (Eds.), *Perceiving, acting, and knowing*. Abingdon-on-Thames: Lawrence Erlbaum Associates.

Gibson, J. J. (1979). *The ecological approach to visual perception*. Boston: Houghton Mifflin.

Günther, F., Dudschig, C., & Kaup, B. (2018). Symbol grounding without direct experience: Do words inherit sensorimotor activation from purely Linguistic context? *Cognitive science, 42,* 336–374.

Humphreys, P. (2004). *Extending ourselves: Computational science, empiricism, and scientific method*. Oxford: Oxford University Press.

Janssen, T. M. V. (2017). Montague semantics. In Zalta, E. N. (Ed.), *The Stanford encyclopedia of philosophy* (Spring 2017 Edition).

Liskov, B., & Zilles, S. (1974). Programming with abstract data types. *ACM Sigplan Notices, 9*(4), 50–59.

McCullogh, W. (1960). *What is a number that a man may know it, and a man, that he may know a number?* Alfred Korzybski Memorial Lecture, Institute of General Semantics, Princeton Club, NY, NY. First published in *General Semantics Bulletin*, No. 26/27, 7-18. Reprinted in www.vordenker.de (Winter Edition 2008/2009) J. Paul (Ed.) http://www.vordenker.de/ggphilosophy/mcculloch_what-is-a-number.pdf.

McCullough, W. (1964). What's in the brain that ink may character? *International congress for logic, methodology and philosophy of science*, Jerusalem, Israel, August 28, 1964, first published in *Embodiments of Mind* (pp. 387–397). MIT Press. Reprinted in www.vordenker.de (Winter Edition 2008/2009) J. Paul (Ed.). http://www.vordenker.de/ggphilosophy/mcculloch_whats-in-the-brain.pdf.

Miyazaki, H. (2013). *The wind rises*. Koganei: Studio Ghibli.

Müller, V. C. (2015). Which symbol grounding problem should we try to solve? *Journal of Experimental & Theoretical Artificial Intelligence, 27*(1), 73–78.

Newell, A. (1980). Physical symbol systems. *Cognitive Science, 4,* 135–183. (Newell includes the following footnote to the paper title. "Herb Simon would be a co-author of this paper, except that he is giving his own paper at this conference. The key ideas are entirely joint".).

Newell, A., & Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM, 19*(3), 113–126.

Norman, D. A. (1988). *The psychology of everyday things*. New York: Basic Books.

Norman, D. A. (2013). *The design of everyday things: Revised and* (expanded ed.). New York City: Doubleday.

Pearl, J. (2000). *Causality: Models, reasoning, and inference*. Cambridge: Cambridge University Press.

Piccinini, G. (2015). *Physical computation: A mechanistic account*. Oxford: Oxford University Press.

Rosen, G. (2017). Abstract objects. In E. N. Zalta (Ed.), *The Stanford encyclopedia of philosophy* (Winter 2017 Edition).

Sony Corporation. (1974). Ad for a Sony sound system. *Radio-Electronics*, p 13. https://www.americanradiohistory.com/Archive-Radio-Electronics/70s/1974/Radio-Electronics-1974-02.pdf.

Taddeo, M., & Floridi, L. (2007). A praxical solution of the symbol grounding problem. *Minds and Machines, 17*(4), 369–389. This paper is reprinted in Floridi, L. (2011) The Philosophy of Information (Oxford, UK: OUP).

Turing, A. M. (1936) On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, 2* (published 1937)*, 42*(1), 230–265. https://doi.org/10.1112/plms/s2-42.1.230. Turing, A. M. (1938). On computable numbers, with an application to the Entscheidungsproblem: A correction. *Proceedings of the London Mathematical Society, 2, 43*(6), 544–6. https://doi.org/10.1112/plms/s2-43.6.544.

Turner, R. (2011). Specification. *Minds and Machines, 21*(2), 135–152.

Turner, R., & Angius, N. (2017). The philosophy of computer science. In E. N. Zalta (Ed.), *The Stanford encyclopedia of philosophy* (Spring 2017 Edition).

The Unicode Consortium. (2017). *The Unicode® Standard Version 10.0—Core specification*. The Unicode Consortium. http://www.unicode.org/versions/Unicode10.0.0/UnicodeStandard-10.0.pdf.

US Copyright Office. Website, FAQ page. https://www.copyright.gov/help/faq/faq-general.html. Accessed September 2, 2017.

van Eijck, J. (2010) A program for computational semantics. Slides for informal presentation. Centre for Mathematics and Computer Science (CWI), Amsterdam, Universities of Utrecht, Amsterdam. https://homepages.cwi.nl/~jve/courses/10/pdfs/APFCS.pdf and http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.297.9166&rep=rep1&type=pdf.

van Eijck, J., & Unger, C. (2010). *Computational semantics with functional programming*. Cambridge: Cambridge University Press.

Wang, H. (1957). A variant to Turing's theory of computing machines. *Journal of the ACM, 4*(1), 63–92.
Wetzel, L. (2009). *Types and tokens: On abstract objects*. Cambridge: MIT Press.
Wetzel, L. (2018). Types and tokens. In E. N. Zalta (Ed.), *The Stanford encyclopedia of philosophy* (Fall 2018 Edition).
Wirth, N. (1971). The programming language Pascal. *Acta Informatica, 1,* 35–63.
Woodward, J. (2003). *Making things happen*, volume 114 of Oxford studies in the philosophy of science. Oxford: Oxford University Press.
Woodward, J. (2014). A functional account of causation; or, a defense of the legitimacy of causal thinking by reference to the only standard that matters—Usefulness (as opposed to metaphysics or agreement with intuitive judgment). *Philosophy of Science, 81*(5), 691–713.