sciendo

STUDIES IN LOGIC, GRAMMAR AND RHETORIC 63 (76) 2020

DOI: 10.2478/slgr-2020-0025

Raymond Turner
University of Essex, UK
e-mail: turnr@essex.ac.uk
ORCID: 0000-0001-6014-4215

COMPUTATIONAL INTENTION*

Abstract. The core entities of computer science include formal languages, specifications, models, programs, implementations, semantic theories, type inference systems, abstract and physical machines. While there are conceptual questions concerning their nature, and in particular ontological ones (Turner 2018), our main focus here will be on the relationships between them. These relationships have an *extensional aspect* that articulates the propositional connection between the two entities, and an *intentional* one that fixes the direction of governance. An analysis of these two aspects will drive our investigation; an investigation that will touch upon some of the central concerns of the philosophy of computer science (Turner 2017).

Keywords: intention, specification, correctness, verification.

Specifications and Programs

Program specifications[1] say what a computer program is intended to do. But we need to distinguish between the propositional content of a "specification", and the intentional act of taking something to be a specification. To explain this remark consider the standard definition of the greatest common divisor.

> A number z is the *greatest common divisor* of two numbers x, y if z divides both and is the biggest one that does.

More formally, the *greatest common divisor* maybe defined by the following predicate calculus expression.

1. $Gcd(x, y, z) \doteq D(z, x) \wedge D(z, y) \wedge (\forall w.(D(w, x) \wedge D(w, y)) \rightarrow w \leq z).$

where $D(z, x)$ states that z divides x. By itself, this is a logical definition pure and simple i.e., it is a description of a relationship between three numbers. Its standard role is mathematical in that it forms a starting point for

mathematical investigation. But as it stands this is not a specification of anything. It makes no reference to anything outside itself. It only does so when an agent takes it to have normative governance over an algorithm or program. For example, it may be taken to have governance over the following program (Fig. 1) written in the WHILE programming language (Hennessy 2020).

```
var x,y,r: Num
begin
read x,y;
r:=x mod y;
while r notequal 0 do
x:=y; y:=r; r:=x mod y
od; write y
end.
```

Fig. 1

This is the *intentional* part of the relationship between definitions and programs. It informs us that the definition is to act as a specification of the program: it determines the nature of the relationship as one of governance.

Notice that it does not spell-out the exact form of the propositional relationship between the two; it does not tell us what it is for a program P to behave in accord with the definition. This is given by the following condition of correctness.

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

Here the program P is semantically understood as a relation between inputs and outputs. This is the statement of accordance or correctness: the input/output behavior of any such program must agree with the definition *Gcd*.

How is (2) established? Presumably, by a formal proof that links the proof theory for the WHILE programming language to the proof theory of the logical language. One such approach is given by Hoare logic (Hoare 1970). Potentially, this delivers a formal proof of (2) for Euclid. So, in this case the relationship is a mathematical one.

In summary, one relationship between logical definitions and programs is that of specification. This has both an intentional aspect (the intentional act of taking the logical definition to have normative governance over the

program) and an extensional one (the conditions of correctness). While this is a toy example, the conceptual points remain intact even when the complexity of the two players (definitions and programs) is increased.

This is our first instance of a relationship between two central computational notions (logical expressions and programs) that is fixed by taking the former to be a specification of the later[2].

Verifying Programs

In practice, programs are seldom proven correct. In practice, programs are subject to experimentation. How exactly? Through the implementation of the WHILE programming language, *Euclid* maybe treated as a physical device that is subject to physical investigation. Any run of the *Euclid* program will generate a physical process that should return some output for given inputs. Under this interpretation, the program *Euclid* is no longer treated as a mathematical object but is replaced by a physical device: the device generated at run time by the implementation. Experimentation then takes the following form: we run the program and see if it behaves as predicted by (1, 2).

However, there are hidden assumptions here. On the face of it, by doing this, we are not testing the physical correlate of *Euclid* but the whole system including the compilers, interpreters, operating systems and hardware. Running the program invokes the whole software and hardware system.

We might attempt to isolate any individual program by taking the supporting implementation and system to be itself correct. While some of this might in principle be proven correct, this cannot apply to the whole system. There will always be some physical residue that has to be verified as correct (Fetzer 1988). But assuming that we have such a verification for the containing system, we may proceed as if we are examining the correctness of the *Euclid* program itself.

The logical assertion is still taken to be the specification of this physical device and to have governance over it. Assertions (2) is then taken to refer to the physical device Π that is generated as the run-time correlate of *Euclid*.

$$2' \qquad \forall x : Num. \forall y : Num \cdot \forall z : Num \cdot \prod(x, y, z) \rightarrow Gcd(x, y, z).$$

However, there is a significant difference: this is an empirical assertion that refers to a physical device and, consequently, cannot be established by math-

ematical proof; it can only be established by experimentation employing the implementation as the tool of investigation.

Depending on how we interpret the program we get different notions of correctness: a mathematical notion and an empirical one. But these different interpretations are fixed by the same intentional attitude: the definition (1) is given normative governance over the program. If we intend to prove that the abstract program satisfies (2) then we shall be engaged in a mathematical activity; if we assume that we are dealing with a physical device generated by the implementation, then we are engaged in an empirical one. While this is a significant difference, both interpretations take the definition (1) as a specification with normative governance. But there is another intentional option.

Theories of Programs

Assume that we do not understand the WHILE language. Imagine that we come across the text of *Euclid* scribbled on a piece of paper, and are intrigued by it. Consequently, we pose the question: "what does it do?". We search online to locate an implementation of the language and run the program on a series of different inputs. Subsequently, we formulate the hypothesis that the program computes the *Gcd*. The definition (1) is now acting not as a specification but as a *theory* of what the program does.

In this scenario, *Euclid* is again treated as an empirical device that is subject to physical investigation, and once again the relationship between (1) and the program is still governed by (2'), but now there is a different justification for it. On this scenario *Gcd* is seen as a *theory* of the program in the sense that it makes predications about the behavior of the program. In other words, we are using the word "theory" in one of the standard senses given to it by the philosophy of science (Winther 2016).

So while the two correctness conditions make the same extensional demands on the relationship between definition and program, there is a significant intentional difference between the specification and theory scenarios. As a theory of the *Euclid* program the assertion (1) can be wrong: if we run the program and it does not satisfy (2') then we will conclude that (1) is not a correct theory. Moreover, as a consequence, we will not change the program *Euclid* as we would have in the specification scenario. The program is the thing that is fixed. If there is disagreement, it is (1), the theory of the program, that is subject to revision.

When there is discord between (2') and *Euclid* different paths are followed.

- In the specification scenario, when (2') fails, we keep (1) but change *Euclid;*
- In the theory scenario, when (2') fails, we keep *Euclid* but change (1).

There is a fundamental difference between the theory and specification scenarios: what governs what is different and, when accordance fails, what gets revised is different. While the extensional relationship of accordance (2') is the same, the intentional aspects are orthogonal to each other.

There is some analogy here with the concepts of illocutionary force and semantic content (Alston 200). In this distinction the semantic content of a sentence, the proposition expressed by it, is independent of any illocutionary uses made of it. For example, the bare proposition may be used as an assertion, a question of an imperative. Likewise, the propositional content of Gcd is independent of its use. It may be employed as a definition, its initial employment, in which case it is employed for mathematical purposes. It maybe used as a specification, in which case it an has engineering function that forms part of the design process for programs. Or it maybe employed as a theory of what a given program does. This adds another dimension to the proposition/illocutionary force duality.

Intentional Shift

The construction of software and hardware systems is complex and the specifications often vague and ill-defined. Nevertheless, the logical relationship between specification and artifact remains the same: specifications have normative force over the construction of the artifact. However, in the bigger enterprise of software development, the path to formulating a specification is itself a complex intentional activity. It involves interaction with clients and stakeholders in order to formulate the requirements of any software or hardware system. It involves a process of building a "theory" of the clients requirements. Such activity might begin with some preliminary theory of the requirements. Focus groups and interviews might be used to abstract such. But whatever techniques are employed, the important point is the nature of the activity: it is a process of theory construction that is tested and revised until there is some stability and agreement. The intention is to get an abstract characterization of the clients demands. If we get it wrong, it does not correctly capture the demands, we revise the theory. Again, this is an empirical enterprise.

As we have seen, theory construction is a very different activity to the process of employing a specification as a normative guide to the construction of a software system; a guide that provides the criteria of correctness and malfunction. However, once stabilized a theory, obtained by an analysis of the clients requirements, maybe used as a specification for the construction of a software system that meets the clients demands. At this point an *intentional shift* occurs: what starts as an empirical proposition, subject to revision, is baptized as a specification that fixes the correctness of any potential system. It is, as Wittgenstein puts it, "hardened into a rule" (Wittgenstein 1978).

> It is as if we had hardened the empirical proposition into a rule. And now we have, not an hypothesis that gets tested by experience, but a paradigm with which experience is compared and judged. And so a new kind of judgement. RFM, VI-22.

What was before a theory of the clients demands, a empirical hypothesis, has been transformed into something that has normative force. Such accounts are of course idealizations where 'idealization' is the process by which scientific models and theories assume facts about the phenomenon being modeled that are strictly false but make theories easier to formulate and explore. When it is determined whether the phenomenon approximates an 'ideal case,' then the model or theory is applied to make a prediction based on that ideal case.

In our case, the process of shifting between one stance and another may continue throughout the software construction process, but the central conceptual point concerns the distinction between theory construction and specification as intentional attitudes, and the fact that the same propositional content can shift its intentional status between theory and specification.

Machines

This distinction between theory construction and specification is important for another topic in the philosophy of computer science, and this concerns the relationships between abstract and physical machines. We may adopt either of the following intentional attitudes to the relationship between abstract and physical machines.

- The abstract machine might be taken as a *representation* of the physical one.

- The abstract machine might be taken as a *specification* of the physical one.

Consider the first. Here the abstract machine must somehow represent the physical device. This is the perspective adopted in (Horsman 2014). It is based upon is the central notion of representation between models and physical systems used in physics (Horsman 2014).

> The key to the interaction between abstract and physical entities in physics is via the representation relation. This is the method by which physical systems are given abstract descriptions: an atom is represented as a wave function, a billiard ball as a point in phase space, a black hole as a metric tensor and so on. That this relation is possible is a prerequisite for physics: without a way of describing objects abstractly, we cannot do science. ... We argue that a 'computer' is a physical system about which we have a set of physical theories from which we derive both the full representation relation and the dynamics.

Representation begins with a physical machine P which has physical states and physical operations linking them. The objective is to represent this with an abstract machine A with corresponding abstract states and operations. The heart of the representation is a function

$$F : A \Rightarrow P$$

that for each (e.g. binary) operation o of the physical machine, associates an operation p of the abstract one such that the following holds.

$$F(p(a, b)) = o(F(a), F(b))$$

In addition, we need to ensure that the physical machine is completely represented in the sense that all its states are represented. Thus, we require the function to be surjective.

All this is in harmony with one version of the so-called *Simple Mapping Account* of physical computation (SMA) (Putnam 1980, Piccinini 2015, Piccinini 2016).

> There is a mapping from the states of the abstract system A to the states of the physical system P, such that the state transitions between the abstract states mirror the state transitions between the physical ones.

This is correctness condition for the abstract machine to *represent* the physical one. Intentionally, the physical device is in charge i.e., if there is a disparity between the two, the abstract machine gives way: it must be modified. It is the physical system that we attempting to represent.

The second intentional stance is that of specification. Characterizing the correctness of a physical artifact relative to its specification poses the question "does a given physical device satisfy a given specification?". If the artifact does not satisfy the specification, we reject the artifact not the specification. Here the representation relation operates in the opposite direction. The objective is to represent or implement the abstract machine with a physical one. Hence the function goes from the physical machine to the abstract one. Consequently, we require a surjective function

$$G : P \Rightarrow A$$

that for each operation p of the abstract machine, there is an operation o of the physical one such that the following holds.

$$G(o(a, b)) = p(G(a), G(b))$$

This is in harmony with the second version of the *Simple Mapping Account*.

> There is a mapping from the states of the physical system P to the states of an abstract system A, such that the state transitions between the physical states mirror the state transitions between the abstract states.

All this ensures that the physical system behaves in a way that is in accord with the abstract one. Here, where there is discord, we change the physical implementation.

These two scenarios are mirror images of each other and are distinguished by the direction of governance. Consider the question "What is a physical computation?" (Piccinini 2015). Which of these two scenarios addresses this question?

Semantic Intention

The λ-calculus was introduced in the 1930s by Alonzo Church (Church 1941) as a way of formalizing the concept of effective computability. We shall use the calculus to further illustrate our notion of intentional stance, but this time applied to the relationship between different mathematical systems. In particular, we employ the different ways of defining the λ-calculus to illustrate how our notion of intentional stance.

The language of the calculus is given by a recursive definition as follows.

$$t ::= x | \lambda x.t | tt$$

The terms of the language are either variables (x), lambda abstracts ($\lambda x.t$) or applications of one term to another (tt').

In standard texts, the calculus is determined or defined by rules of computation, the central one being the rule of *β-reduction* given as follows, where $\succeq$ indicates reduction. The rule allows the replacement of the left hand side ($(\lambda x.t)s$) by the right hand side ($t[s/x]$) where the latter denotes the substitution of the term s for the variable x in the term $.t$

$$(\lambda x.t)s \succeq t[z/x]$$

Computation proceeds by using the *β-rule* to reduce lambda terms. When such computations terminate, we reach the result of the computation – its normal form. This is an idealized version of a programming language and its underlying concept of computation. Some theoretical computer scientists and logicians would see this as defining the actual calculus i.e., the rules define the calculus as a mathematical system.

Others do not. Instead, they take set-theoretic models to fix matters, where such models are by now various. There are even models constituted by the recursively enumerable sets. But the original lattice theoretic model was based upon a complete lattice that was isomorphic to its own continuous function space (Scott 1980).

$$D \cong [D \Rightarrow D]$$

This enabled the semantic account of the calculus to be given as follows – where e is a function that maps variables to the domain of the model, and where $e[d/x]$ is the modification of e that maps x to d.

$$[\![x]\!]_e = e(x)$$

$$[\![\lambda x.t]\!]_e = f \quad \text{where} \quad f(d) = [\![t]\!]_{e[d/x]}$$

$$[\![tt']\!]_e = [\![t]\!]_e([\![t']\!]_e)$$

This semantic definition makes sense because under the isomorphism the function f may be considered as an element of the domain, and for application any element of the domain may be considered as a function so that it may be applied to any other element. However, the technical details need not detain us. The question we need to ask is "what governs what?"

From the perspective of the calculus itself, any model must make the rules of reduction sound in the sense that, under the semantic interpretation, each side must denote the same element of the domain of the model.

It would not be a model if it did not respect the rules of the formal calculus. This is an operational view that puts the formal calculus in the driving seat.

Dana Scott, the inventor of the domain theoretic models, seems to have taken the models as having semantic governance (Scott 1980). Seemingly, the calculus cannot stand as a mathematical theory without a set-theoretic model. The existence of such a model is taken as a hygiene test for such formal theories. They guarantee its consistency and, without the latter[3], they would need to be modified. The models are the source of meaningfulness, and the language of the calculus is only a way of talking about the real objects that occupy the model. This puts the models in charge. Underlying this is the belief that set-theory provides the ultimate foundations for mathematics. This is often coupled with a realist view of set-theory (Turner 2007).

In this case, the underlying intentional stance reflects a deep rooted philosophical difference.

Dominant and Submissive

Our relationships consist of dominant/submissive pairings. All the examples chosen have been taken from the computational arena. But the phenomenon is much wider.

The *function* and the *structure* of technical artifacts, from the philosophy of technology (Kroes 1998, Kroes 2012), is an instance of this dominant/submissive pairing where the functional definition of the artifact has normative governance over its structural description. The function lays out the conditions of correctness; it is a specification of the artifact. On the other hand, the structural description describes the artifact as a physical object.

Dominant/submissive pairings can also be found throughout mathematics and logic. One example concerns the semantic definition of first-order predicate logic coupled with any axiom/proof system for the later. Here the truth-conditional semantics for predicate logic is dominant over the proof theory. The correctness conditions are provided by the soundness and completeness results. If the axiom system does not satisfy these demands, it is a candidate for change.

Other instances come from the philosophy of science (Frigg 2018). We have already discussed a very special instance of theory construction. But the notion of intentional stance has much wider application to the philosophy of science where the construction of theories and models is at the heart of matters (Winther 2016, Frigg 2018).

All of these are instances of our dominant/submissive paradigm. A little reflection should locate many others. Indeed, such intentional concerns have played little role in the philosophies of mathematics, science and engineering.

NOTES

* Written as part of "PROGRAMME" ANR project: What is a program? Historical and philosophical perspectives.

[1] (Turner 2015)

[2] This analysis is inspired by the two aspects on intention mentioned by David Pears in his essay on the private language argument in (Pears 2006).

[3] In the case of the Lambda Calculus, the Church-Rosser theorem provides a form of consistency proof.

REFERENCES

Alston, W. P. (2000). *Illocutionary Acts and Sentence Meaning*. Ithaca: Cornell University Press.

Church, A. (1941). *The Calculi of Lambda-Conversion* (ISBN 978-0-691-08394-0) [1]

Colburn, T. (2000). *Philosophy and Computer Science*. M.E. Sharp Publishers. New & rk, London.

(1988). Fetzer. J.H. Program Verification: the very idea. Communications of the ACM. Volume 3:9.

Frigg, R. & Hartmann, S. (2018). Models in Science. The Stanford Encyclopedia of Philosophy (Summer 2018 Edition), Edward N. Zalta (ed.), URL = https://plato.stanford.edu/archives/sum2018/entries/models-science/.

Hoare, C.A. (1969). An Axiomatic Basis for computer programming. Communications of the ACM, Volume 12, Issue 10.

Horsman, C., Stepney, S., Wagner, R. & Kendon, V. (2014). *When does a physical system compute?* Proc. R. Soc. A September 8.

(1988). Kroes, P. Technological explanations: the relation between structure and function of technological objects. Techné: Research in Philosophy and Technology 3 (3), 124–134.

Kroes, P. (2012). Technical Artifacts: Creations of Mind and Matter: A Philosophy of Engineering Design. Springer.

Pears, D. (2006). *Paradox and Platitude in Wittgenstein's Philosophy*. Oxford: Oxford University Press.

Piccinini, G. (2015). Computation in Physical Systems. *The Stanford Encyclopedia of Philosophy* (Summer 2015 Edition).

Piccinini, G. (2016). *Physical Computation: A Mechanistic Account*. Oxford.

Putnam, H. (1980). Minds and Machines. In *Dimensions of Mind: A Symposium*. S. Hook (ed.), New York: Collier. pp. 138–164.

Rapaport, W. J. (2020). Philosophy of Computer Science. https://cse.buffalo.edu/~rapaport/Papers/phics.pdf

Scott D.S. (1980). Lambda calculus: Some models, some philosophy Barwise, et al. (Eds.), The Kleene Symposium, Studies in Logic, 101, North-Holland, pp. 381–421.

Turner, R. (2007). Understanding Programming Languages. Minds and Machines 17(2): 203–216.

Turner, R. & Angius, N. (2017). The Philosophy of Computer Science. *The Stanford Encyclopaedia of Philosophy* (Spring 2017 Edition).

Turner, R. (2018). *Computational Artifacts. Towards a Philosophy of Computer Science*. Springer.

Turner, R. (2011). Specification. Minds and Machines 21 (2): 135-152 .

Turner, R. (2020). Correctness, Explanation and Intention. DOI: 10.1007/978-3-030-22996-2_6. In Computing with Foresight and Industry. Springer.

Winther, Rasmus Grønfeldt. (2016). The Structure of Scientific Theories. The Stanford Encyclopedia of Philosophy (Winter 2016 Edition), Edward N. Zalta (ed.), URL = https://plato.stanford.edu/archives/win2016/entries/structure-scientific-theories/.

Wittgenstein, L. (1978). *Remarks on the Foundations of Mathematics*. G. H. von Wright, R. Rhees, and G. E. M. Anscombe (eds.). Oxford.

Hennessy, M. https://www.scss.tcd.ie/Matthew.Hennessy/splexternal2015/notes/WhileSlides2to1.pdf