philosophies

MDPI

Article

Meaning and Reference in Programming Languages

Nicola Angius

Department of Cognitive Science, University of Messina, 98121 Messina, Italy; nicola.angius@unime.it

Abstract: This paper provides an analysis of the meaning–reference problem, as traditionally advanced in the philosophy of language, for imperative programming languages. Addressing the problem is relevant for the philosophy of computer science in the context of the debate of whether simulative programs can be considered representational theories of their simulated systems. This paper challenges the thesis that, since programming languages have semantics, they also have reference, presumably to the external world. First, denotational, operational, game theoretic, and axiomatic semantics are epistemologically analysed to provide a theory of meaning for high-level language programs. It is argued that programming language semantics can be seen, to some extent, under the light of Fregean descriptivism, thereby rejecting the view that it is reference that determines meaning. Secondly, descriptivist, causal, character, and intentionalist theories of reference are examined to argue that meaning is not sufficient for programming languages to determine reference, thereby rejecting the view that meaning determines reference. It is argued that it is the intentions of agents that allow programming language semantics to identify referents. The paper concludes that if the semantics of programming languages may determine their reference, the latter is not to the external world but to the internal operations of the implementation executing them. This does not prevent simulative programs from representing empirical systems, but they do so not on the basis of programming language semantics.

Keywords: philosophy of computer science; operational semantics; denotational semantics; meaning; reference; externalist view of programs; programs as theories

check for updates

Academic Editors: Raymond Turner and Henri Stephanou

Received: 18 November 2024
Revised: 25 March 2025
Accepted: 26 March 2025
Published: 1 April 2025

Citation: Angius, N. Meaning and Reference in Programming Languages. *Philosophies* **2025**, *10*, 40. https://doi.org/10.3390/philosophies10020040

Correction Statement: This article has been republished with a minor change. The change does not affect the scientific content of the article and further details are available within the backmatter of the website version of this article.

Copyright: © 2025 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The philosophy of computer science has developed as an attempt to provide an analysis of computer science under the light of traditional philosophical disciplines such as ontology, epistemology, and ethics [1,2]. Ontological examinations involved investigations into, for instance, the nature of computational systems [3], algorithms [4], or programs [5,6]. The epistemology of software verification and testing [7,8], malfunctioning [9], modelling, and abstraction [10,11] helped in understanding the epistemological status and the foundations of computing as a discipline [12,13]. Computer ethics focuses on compelling ethical–legal issues concerning, among others, the moral responsibility of programmers [14], the intellectual property rights of software [15], or the moral agency of computing systems [16].

One important area of philosophy is missing in this picture, namely the *philosophy of language* At the same time, informal and formal languages play a pivotal role in software development and verification, from specification languages through high-level programming languages to specification and verification logic. The formal languages of computer science are defined by syntax, semantics, and their relationship, which are well examined in the philosophy of language and are the focus of this Special Issue on "Semantics and Computation".[1]

This paper aims to contribute to the analysis of the semantics of computational languages by considering the *meaning–reference problem* for imperative, high-level programming languages. The problem is a crucial one in the philosophy of language and, as originally put forward in the nineteenth century by the German logician Gottlob Frege [17], it is the problem of determining whether the meaning of a language expression is determined by what the expression refers to in the external world, or, vice versa, whether it is meaning that determines reference, or even if meaning and reference are two linguistically and cognitive independent aspects of language. Over the years, philosophers of language advanced many different *theories of meaning* [18] and *theories of reference* [19], tackling the meaning/reference problem in divergent ways.[2]

In the context of programming languages, the meaning–reference problem becomes the question of whether the meaning associated with the constructs of a language enables that language to refer to something.[3] Indeed, high-level programming languages are defined by syntax, defining the construction rules for instructions, and semantics, which are associated with each type of instruction and meaning. Three related problems are at stake here: whether programming languages are referential, if they are in virtue of their semantics, and what kind of system they refer to.

The question of whether programming languages are referential is relevant to another important debate in the philosophy of computer science, i.e., the problem of whether computer programs are *theories* of an empirical system. The programs-as-theories thesis was initially advanced in the context of the philosophy of artificial intelligence (AI) by Allen Newell and Herbert Simon [20] and subsequently discussed in the philosophy of cognitive science [21,22]. In its original formulation, it is the thesis that simulative programs in AI can be considered as theories of the mind.

Moor [23] emphasises how considering simulative programs as theories of their simulated system is one of the myths of computer science. Programs can be understood best as models rather than theories. Theories involve predictive and explanatory law-like statements that programs do not include, while both programs and models may be able to represent any given target system.[4] The author nonetheless highlights how programs can be understood as models provided that isomorphisms can be established between a program and a represented system, in the same way scientific models do with their target empirical systems. The question left open by Moore can be paraphrased by asking whether programs, *qua* linguistic entities, are referential or, in other words, whether they are able to refer to, and therefore represent, a system.

Answers to this problem have divided *internalist* and *externalist* views of computation [1]. According to the internalist view, programs are internal, narrow, local, and *syntactic*; in other words, computation is understood as a manipulation of symbols and, at the physical level, it concerns the operation of the physical machine implementing the program.[5] Typically, the internalist approach is in favour of the formal verification of correctness: all that is needed to prove correctness is to demonstrate that a program is a model of a specification expressing its desired functionalities; reference to the external word is not needed to understand and evaluate program correctness.[6]

In contrast, the externalist view sees programs as *directly* referring to the world. Programs are, in this case, understood as wide, global, and *semantic*. Those, such as Newell and Simon [20], arguing that programs are theories of the mind or of other simulated systems, support the externalist view of programs. Holders of the externalist view are typically critical towards formal verification: program correctness is not just about the formal relationship between a specification and a program code, in that a machine implementing a program satisfying its specification may still fail in achieving the desired goals due to faulty hardware or a deficiency in the interaction with the environment. In other

words, correctness is all about making an implemented program behave in its environment as intended.[7] Finally, according to the externalist view, programs are referential in that they are semantic: the semantics of a programming language provide a reference for the programming language.

Proponents of the externalist view of programs assume the semantics of programming languages to be referential. Whether semantics are or are not referential has also been a crucial problem in the philosophy of language in connection with the so-called meaning/reference problem [25]. Originally put forward by Gottlob Frege in [17], it is the problem of determining whether the expressions of natural languages have meaning by virtue of their being able to refer to some external object in the world or not. In particular, what the externalist view of programs seems to assume is the so-called "Fregean slogan", according to which meaning determines reference [18]. The aim of this paper is to examine whether this is the case also for programming languages.

To this aim, Section 2 introduces to the four main programming language semantics, namely operational, denotational, game theoretic, and axiomatic semantics. Section 3 considers the main theories of meaning developed in the philosophy of language to identify which of them better reflects the usage of programming language semantics. Section 4 examines the theories of reference in philosophy of language, establishing a relation between semantics and reference to consider whether they apply to programming languages. Finally, Section 5 answers the starting question, namely whether Fregean slogan holds true for programming languages.

2. The Semantics of Programming Languages

Before defining the notions of meaning and reference for programming languages, let us briefly recall what the semantics of programming languages amount to. As stated above, every programming language is associated, in the first place, with a set of syntactic rules defining the constructs of the language. For the sake of simplicity, let us consider a pseudo-imperative language defined by an assignment instruction, a selection instruction, a *while* loop, and a sequential composition. Syntax defines each construct and the elements appearing there. The assignment instruction is defined as the construct,

$$a := E,$$

requiring that the value 'E' be assigned to the variable 'a', where 'a' corresponds to a memory location in the machine running the program. 'E' can be a constant, a variable, or an arithmetic expression made up of variables and constants.

The same goes for the other constructs. A syntactic rule may define the selection instruction as the conditional construct:

$$if\ E\ then\ I\ else\ I',$$

requiring that instruction I be executed if a given condition E is true or a different instruction I' if the condition is false. By the same token, the *while* loop can be defined as the construct:

$$while\ E\ do\ I,$$

evaluating condition E and, as long as it holds true, requesting instruction I to be executed. Finally, sequential composition can be syntactically defined as follows:

$$I; I',$$

allowing two or more instructions to be composed in a single code line.

Once syntax is fixed, semantics associate a meaning in different ways to each construct, depending on the kind of semantics at stake. In the following, the focus is restricted to the main programming language semantics considered in computer science, namely operational, denotational, game-theoretic, and axiomatic semantics. These are called *static semantics*, in that program execution is not required; they aid formal verification properly in that they associate a meaning to the program code. So-called dynamic semantics, by managing runtime errors, concurrency, or runtime resource management, will not be considered here in that this study is more concerned with how instructions are executed rather than directly with their meaning.

2.1. Operational Semantics

The main idea underpinning operational semantics is that the execution of a given instruction determines a state change in the system [26,27]. This is often expressed as $< i, s >\Downarrow s'$, stating that instruction i, executed in state s, brings the system into state s'.[8] To each syntactical construct, operational semantics associate a corresponding semantic rule expressing the system state change induced by the execution of the instruction.

For instance, the operational semantic rule associated with the assignment instruction is as follows:

$$\frac{(Val(E, s) = v)}{(< a := E, s >\Downarrow Update[a, s, v]).}$$

Specifying that given v the arithmetic value of expression E in state s (premise), the execution of the assignment instruction $x := E$ in state s brings the system into a new state, identical to s, except the fact that the machine updates s by assigning value v to a (conclusion).[9] Ultimately, the meaning of the assignment instruction is an update operation of the machine, namely $Update[a, s, v]$.

The semantic rule for the conditional instruction corresponds to the following:

$$\frac{< E, s >\Downarrow true, < I, s >\Downarrow s'}{< if\ E\ then\ I\ else\ I', s >\Downarrow s',}$$

$$\frac{< E, s >\Downarrow false, < I', s >\Downarrow s'}{< if\ E\ then\ I\ else\ I', s >\Downarrow s'.}$$

The first rule states that if condition E is true in state s, and if executing I in state s brings to state s', then the execution of the conditional instruction in state s brings state s'; the second rule goes in the same way but assumes that condition E in state s is false and, therefore, instruction I' is executed in state s, bringing state s'.

The semantic rule associated with the *while* loop is divided into two cases as well, depending on whether condition E in state s is true or false:

$$\frac{< E, s >\Downarrow false}{< while\ E\ do\ I, s >\Downarrow s,}$$

$$\frac{< E, s >\Downarrow true, < I, s >\Downarrow s', < while\ E\ do\ I, s' >\Downarrow s''}{< while\ E\ do\ I, s >\Downarrow s''.}$$

Finally, the semantic rule for the sequential composition states that if instruction I executed in state s leads to state s', and I' executed in state s' leads to state s'', then executing the composition of the two instructions in state s brings the machine to state s'':

$$\frac{< I, s >\Downarrow s', < I', s' >\Downarrow s''}{< I; I', s >\Downarrow s''.}$$

As stated above, the semantics of programming languages are conceived as a means to prove the formal correctness of programs. In the case of operational semantics, given a program, the semantic rules allow one to build a *state transition system* representing the state changes in the implementing machine when instructions are executed. A state transition system is, in other words, an *abstract machine*. Correctness can be proved by formalising the set of specifications in an appropriate logic, for instance, *temporal logic*, and a *model-checking* algorithm checks whether each formula holds true in the model.[10]

2.2. Denotational Semantics

In denotational semantics [29–31] the syntactical constructs of a programming language are interpreted as elements of mathematical structures called *domains*. Such interpretations are models of programming languages: a domain is a partially ordered set and a model is a family of partially ordered sets, each interpreting types of the language, such as natural numbers, variables, Boolean constants, or functions.[11] Generally speaking, denotational semantics are defined by a set of domains, providing meaning to program constructs and a set of functions mapping syntactical constructs to their meaning. In other words, denotational semantics interpret a program as a set of functions from domain to domain.

Given a program, the functions of a denotational semantics map for each instruction, from input states to output states, result from the execution of the given instruction. Since it may well be the case that, for a given instruction, the program does not terminate and there is no output state, denotational semantic functions are partial. A straightforward way to present denotational semantics is to introduce a partial equality relation $\simeq$ between input and output states for the given syntactical construct, as conducted in [3]. The partial equality relation for the assignment instruction is as follows:

$$||a := E||s \simeq Update(a, ||E||s, v),$$

associating state s, where the assignment instruction is executed, to the state resulting after the $Update$ operation. The partial equality relations for, respectively, the selection instruction, the *while* loop, and the sequential composition can be defined in a similar way as follows:

$$||if\ E\ then\ I\ else\ I'||s \simeq \{||I||s\ if\ ||E||s\ is\ true;\ ||I'||s\ if\ ||E||s\ is\ false\},$$

$$||while\ E\ do\ I||s \simeq \{s\ if\ ||E||s\ is\ false;\ ||while\ E\ do\ I||(||P||s)\ if\ ||E||s\ is\ true\},$$

$$||I; I'||s \simeq ||I'||\ (||I||)s.$$

Proving program correctness through denotational semantics requires, again, that program specifications be expressed in some formal language which, as Hoare's logic suggests [32], specifies the program's pre- and post-conditions. Correctness is proved by showing that the denotations mapped by the semantic functions satisfy those pre- and post-conditions.[12]

2.3. Game Theoretic Semantics

Both operational and denotational semantics induce an equivalence relation over programs. From an operational point of view, two programs P and P' can be said to be equivalent, $P \equiv_O P'$, in case $< P, s > \Downarrow s' \leftrightarrow < P', s > \Downarrow s'$, i.e., in the case when they are both launched in a given machine state s, their execution leads to the same state s'. Denotational semantics define the equivalence $P \equiv_D P'$ as the identity relation $M(P) = M(P')$, that is, as an identity between the semantic interpretation, or model $M(P)$, of program P and the model $M(P')$ of program P'. According to the *full abstraction property* [33], denotational

semantics are *fully abstract* with respect to an operational one if the relations $P \equiv_O P'$ and $P \equiv_D P'$ agree. In other words, full abstraction is achieved when denotational models are able to exhaustively encompass all the behaviours of the abstract machine for the involved program. This amounts to saying that, on the one hand, two denotationally equivalent programs should be indistinguishable from an observational point of view and, on the other hand, that all observable behaviours, as represented operationally, should also be included in the denotational model. Unfortunately, fully abstract denotational models can hardly be provided for programming languages; this is known as the *full abstraction problem*. The problem is related to the difficulty of making the denotational model so precise so as to include all the needed details of the operational model.

The failure to prove full abstraction for denotational semantics led to game theoretic semantics [34,35], which can be seen as a version of denotational semantics that is indeed able to induce the same equivalence as the operational one. In other words, game semantics are able to provide a fully abstract model with respect to operational semantics.

The main idea at the basis of game theoretic semantics is that computation can be understood as a dialogue between a computing agent and the environment wherein the former operates. Game theoretic semantics understand the meaning of a program in terms of the observable behaviour of the computing agent, wherein interactions between the agent and the environment play a fundamental role [36]. This kind of semantics was inspired by the work of Jaakko Hintikka in game theoretic semantics for logic [37,38], wherein a game is defined by a set of moves, two actors usually called the proponent and opponent, and a set of positions, that is, a series of moves each characterised by an alternation of moves of the opponent and proponent, starting with the opponent. The proponent proves the truth of a statement; the opponent attempts to challenge the proof.

In the context of programming language semantics, the proponent attempts to prove the correctness of the program, and the opponent attempts to prove incorrectness by showing counterexamples. Moves within a position can be understood as strategies for the proponent's actions in response to the opponent's movements. A game then takes the form of a formal structure representing the dialogue game, wherein the proponent answers questions advanced by the opponent. Game theoretic semantics are used to prove program correctness by showing the winning strategy of the proponent.

2.4. Axiomatic Semantics

Axiomatic semantics [39] aim to define the meaning of a program in terms of axioms, i.e., rules, allowing the formal proof of the properties of the program. As with other semantics, axiomatic semantics were conceived in relation to the correctness problem: the original aim was to logically derive the property formalised in a specification from the axioms that define the program [32]. In this case, each program construct is associated with an axiom; an entire program can thus be represented as a set of axioms. Rules of this kind have the form of what is known as a *Hoare's triple* $\{\phi\}i\{\psi\}$, stating that if a certain precondition $\{\phi\}$ holds true at a given state, the execution of instruction i leads to a successor state wherein postcondition $\{\psi\}$ holds true. Hoare's rule for the assignment instruction is as follows:

$$\{\phi[E/x]\}a := E\{\phi\}.$$

This means that the postcondition holds true if the execution of $a := E$ has the same precondition, except E was substituted for variable x. The semantic rule for the conditional statement is as follows:

$$\frac{\{E \wedge \phi\} I \{\psi\}, \{\neg E \wedge \phi\} I' \{\psi\}}{\{\phi\} if\ E\ then\ I\ else\ I' \{\psi\},}$$

stating that the postcondition of both branches of the conditional statement is also a postcondition of the whole statement. The *while* loop axiom

$$\frac{\{E \wedge \phi\} I \{\phi\}}{\{\phi\} while\ E\ do\ I \{\neg E \wedge \phi\},}$$

specifies that if ϕ remains true when E is true and I is executed, then ϕ remains true throughout the loop. Condition ϕ is called the *invariant* of the loop. Finally, the rule for the sequential composition is as follows:

$$\frac{\{\phi\} I \{\psi\}, \{\psi\} I' \{\chi\}}{\{\phi\} I; I' \{\chi\}.}$$

Simply stated, if the postcondition of the first instruction is also the precondition of the second instruction, then, given the precondition of the first instruction, the composition of the two instructions leads to a state where the postcondition of the second instruction holds true.

Specifications can be formalised using Hoares' triple as well, requiring, given a precondition, that *must* be the postcondition.[13] Briefly speaking, once a program has been formalised as a set of Hoare's triple axioms, Hoare's triple specification is treated as a theorem to be derived by the axioms. To this aim, the precondition of the specification is taken as the precondition of the first program instruction, and whether the postcondition of the last program instruction corresponds to the postcondition required by the specification is checked.

As a concluding remark on programming language semantics, only game theoretic semantics and operational semantics can be proven to be equivalent via full abstraction. However, operational semantics are considered to have a "*definitional priority*" over other kinds of semantics, meaning that denotation and axiomatic semantics must agree with operational semantics. According to Turner [3], such a definitional priority has a historical motivation since both Turing machines and the λ-calculus were originally defined using operational semantics. Denotational semantics can be seen as having a more theoretical role, being better suited for exploring the mathematical properties of programs. Axiomatic semantics were conceived to directly enable formal proof of correctness. The definitional priority principle indicates that operational rules are those that better convey the meaning of programming languages by showing how computations are actually carried out. This paper endorses the definitional priority principle by focusing on operational semantics and considering the other kinds of semantics adhering to it.

The next section will now go through the main theories of meaning that have been historically discussed in the philosophy of language to see whether there are any that comply with the semantics of the programming languages presented.

3. Theories of Meaning for Programming Languages

In the philosophy of language, a theory of meaning is a theory associating meaning with the elements of a natural language. Historically, the first attempt to provide meaning to the expressions of a language was by Gottlob Frege and his *Begriffsschrift* [40]. Indeed, a semantic theory was also conceived for logical statements; as is known, the semantics of a logical system can be provided in terms of *model theory* [41], initially formalised by Alfred Tarski [42]. In model theory, the semantics of predicate logic are provided by introducing a world and sets of functions mapping constants and predicates to (sets of) elements of the world. Semantics are presented in terms of truth conditions, so for the meaning of the

sentence, $P(a)$ is its truth condition; $P(a)$ is true if the object in the world corresponding to constant a is an element of the set corresponding to predicate P and false otherwise. The meaning of constant a is simply the world object it is mapped onto. Transposed to natural language, this is tantamount to saying that the meaning of a sentence is its truth condition, and the meaning of an element of a language is its contribution to determining the truth value of the sentence it compares to. The meaning of the sentence 'Antony is a physician' is its truth condition, and the meaning of the term 'Antony' is the person in the real world and what the term is associated with (that is, Antony). This is to say that meaning is provided by reference: in meaning, there is not anything more than reference, a position that has been widely developed for natural languages by Donald Davidson [43].

The thesis that a theory of reference is all one needs to provide semantics works well with formal languages, wherein there are no indexicals, where usually there are no different sentences corresponding to the same proposition, and where a formal world can always be defined to provide meaning. Indeed, among the four kinds of programming language semantics, denotational semantics are the ones that provide meaning to a language in this proper way. Denotational semantics consider programming languages as mathematical languages, and meaning is provided in terms of a model, that is, a set of domains, each interpreting a language type. One should be careful to notice here that the model providing meaning to the language is a formal model representing the state change of an abstract machine. The assignment instruction to recall the first case is associated with a function from a machine state to the same state wherein variable a has been assigned value E. It follows that the model providing meaning to the language is not the outside world, the world any machine implementing the program would interact with; it is a formal model providing a set-theoretic representation of an abstract machine for the program. Also, given the assignment instruction, different semantics, i.e., meanings, can be associated with it beyond the denotational one, all referring, in different guises, to that abstract machine operation. Therefore, there seems to be a many-to-one relation between meaning and reference.

Frege [17] himself famously argued that reference is not sufficient to determine the meaning of a language expression, and different meanings may correspond to the same referent. His well-known example shows that the expressions 'the morning star' and 'the evening star' denote the very same entity in the external world, namely the planet Venus; nonetheless, anybody would agree that the meaning of the two expressions is somehow different. Since the two expressions have the same reference, any two sentences identical except for the two expressions would have the same truth value, such as 'the morning star is the biggest star' and 'the evening star is the biggest star', which have the same meaning according to the referential theory of meaning. Nonetheless, the German logician maintains that the difference in meaning between the two expressions is due to a difference in their content, expressed by the two propositions they convey. In contrast, there are cases of different sentences, such as 'Mary is blonde' and 'Mary has got blond hair', which have the same content, same reference, and same truth value.

The conclusion is that, on the one hand, two expressions having the same reference can convey different propositions and thus have different meanings; on the other hand, two expressions having the same meaning cannot have different references. In this exact sense, Frege states that meaning determines or fixes reference. Given the content of an expression, this will determine its reference.

According to the *propositional theory of meaning*, expressions of a natural language are associated with entities, notably propositions, that determine their meaning. But what are these kinds of entities? Following *Millian–Russellian semantics*, those entities are structured propositions defined by specific elements such as objects, properties, and relations [18]. So,

for instance, a name in a sentence designates an object, an attribute or a property of objects, or a predicate of a relation.[14]

The analyzed programming language semantics associate formal entities with language constructs, and these entities are structured entities, such as operational rules, axioms, or game strategies; also, some of the language elements, such as variables, constants, and expressions, can be associated with formal objects, such as memory locations, numbers, or truth values. However, it remains true that instructions in a programming language are not declarations, and there are elements of the language, say the assignment :=, or the '*while*' symbol, that cannot be associated with single elements of a structured entity.

In keeping with Fregean semantics, the content of a sentence is still a structured entity, but such an entity is not defined by objects, properties, and relations. As the planet Venus example shows, the content of an expression is a *mode* of presentation of an object to a subject. Such a mode of presenting or thinking of objects is called Frege *sense*. Fregean senses are objective contents of language expressions that allow one to determine their referents. In the case of names, sense is presented in terms of a *definite description* which, such as 'the morning star' or 'the current president of the United States', specifies properties or conditions of the referent and thus allows one to single out the referent. Given a referent, such as the planet Venus, this may satisfy different conditions or properties, and this is why it can be referred to by using different definite descriptions such as 'the morning star' or 'the evening star'. This position is known as *Fregean descriptivism* [18].[15]

Beyond the different philosophical problems associated with Fregean descriptivism (such as Kripke's modal argument [44]), one aspect of this theory is useful to understand the meaning of programming languages: the different analysed semantics provide different ways or modes (Fregean senses) of presenting the meaning of programming language constructs. Given, say, the assignment instruction, its meaning depends on a certain way of presenting the operation of an abstract machine, which, in turn, represents the update operation of a memory location value. The update operation of the abstract machine can be presented by means of operational semantics, specifying that if a given variable in a given state has a certain value, the execution of the assignment results in the same machine state but such that the variable has a new value. The same abstract machine operation is presented by denotational semantics but in a different guise, namely in terms of functions from machine states to machine states. A yet different way of presenting the update machine operation is provided by game-theoretic semantics, which sees the update operation as a step-strategy in a game between an agent and the environment trying to prove the correctness of a program. Finally, axiomatic semantics specify that the abstract machine remains in the same state but updates the value of the involved variable and specifies that the postcondition is the same precondition.[16]

To see if the Fregean slogan is valid for the semantics of programming languages, one should examine whether different senses associated with the program constructs determine reference, i.e., whether the constructs refer by virtue of their meaning. To do so, a preliminary question is whether there is a theory of reference that programming languages adhere to. Let us see.[17]

4. Theories of Reference for Programming Languages

One may identify as many as four theories of reference in the philosophy of language [19]. The first one, the *descriptivist model*, is a referential counterpart of the descriptivist theory of meaning provided in the previous section. In the Millian–Russellian guise, the descriptivist theory maintains that sentences express propositions which are structured entities wherein each term or substructure is referential and contributes to the meaning of the whole sentence. So, for instance, the reference of the sentence 'Aristotle is a philosopher'

is its truth value, determined by the predicate 'is a philosopher', defining a property for the object referred to by 'Aristotle', the philosopher born in Stagira in 384 B.C. Identifying referents is essential to define the truth value of sentences.

As noted above in Section 3, even though expressions of programming languages are structured and composed of distinct constructs, each constituted by distinct terms such as variables, Booleans, constants, and commands, semantics do not directly associate any of them with external objects. Semantics rather associate program expressions with formal entities, such as states and transitions, sets and functions, proof rules, or game strategies.

Fregean descriptivism still sees sentences as expressing structured propositions whose constituents, however, are not external objects and properties thereof but rather senses. The sense of an expression is not directly its referent; according to Frege, senses and reference must be distinguished. It was recalled above how the sense of, say, a name is a mode of presentation of the referent to a human subject and how senses are definite descriptions, with regard to propositions of the form 'the morning star', that allow one to determine the unique referent satisfying the description. This is at the core of the Fregean slogan, which is that sense determines reference.

Section 3 argued that programming language semantics comply, to a certain extent, with Fregean descriptivism: each of the examined semantics associates constructs to entities which have some implicit propositional form. Consider again the assignment instruction $a := E$; operational semantics associate the instruction with an operational rule which can be paraphrased by saying that the meaning of the assignment instruction is the state transition such that the new state is identical to the previous one except for the fact that value E has been assigned to a. Similar considerations were extended to the other kind of semantics, which present in different ways (senses) the same propositional content.

The question becomes whether the propositional entities associated with the construct of a programming language permit one to pick out a referent in the external world, such as for Fregean descriptions. This is a known problem in the philosophy of language. The limits of the descriptivist theory of reference have been discussed in connection with definite descriptions with no referents, such as the known example "The King of France is bald" [46], with names having more than one bearer [47], with definite descriptions corresponding to multiple referents [48], and with definite descriptions which are false [49].

All the semantics introduced in Section 2 leverage the notion of the machine state or machine state transition, that is, of an abstract machine.[18] Abstract machines are a formalism that enables one to reason about the property of a program; again, the aim of programming language semantics is to aid verification and correctness evaluation. Meaning to the constructs of a programming language is then provided in terms of the operations of an abstract machine (presented in different ways by different semantics). Let us ask whether this suffices to choose a referent. One should note that the kind of referent an abstract machine may be associated with is the physical machine implementing it, a point already emphasised by [50]. Even though abstract machines may also include variables representing elements of the operating environment and users, they mainly represent the computation that any implementing machine must realise.

Abstract machines do not correspond to a single implementing machine. States of an abstract machine are indeed abstract; that is, they are often *macro-states* that can be mapped to a set of machine states corresponding as a whole to the macro-state. Macro-states are obtained by introducing a set of *macro-variables*, each corresponding to a set of program variables, inducing a collapse of the abstract machine states and transitions. Such a collapse is often required to reduce the state space of the abstract machine and, with it, the computational resources the search algorithm needs to travel it [51]. Data abstraction is also known to generate spurious paths in the abstract machine, that is, paths

not corresponding to any execution of the physical machine [52]. Also, the abstract nature of state transition systems makes them *multi-realisable*: an abstract machine corresponds to a multiplicity of different physical machines, each unpacking macro-sates and transitions in different ways. Multiple realisability characterises not just the abstract machine level but all ontological levels defining a computational system [2]. Accordingly, the meaning associated with programming language instructions does not suffice to individuate a physical machine as its external referent. These further considerations lead us to conclude that there is no many-to-one correspondence between senses and reference, as there is in Fregean descriptivism, but rather a many-to-many relation.

The multiple realisability of abstract machines is an equivalent of the case of proper names having multiple bearers, that is, physical people named in the same way. According to [53], a potential solution to the multiple bearer problem is to treat names as *indexicals*. The latter are not rigid designators [44]: they are terms that do not always refer to the same entity but such that their referents change depending on the context of the utterance of the sentence of interest. Speakers, hearers, times, and places contribute to defining the relevant context of utterance. Examples of indexicals include 'I', 'you', 'here', or 'tomorrow'. If it is me saying 'I', the expression refers to myself or, in general, to whoever is pronouncing the pronoun.

A second theory of reference is the so-called *character model*, related to the work of David Kaplan [54]. According to Kaplan, whereas names have content, indexicals have a character, which is to be understood as some sort of function or rule, mapping from contexts of utterance to referents. The indexical 'I' has a referent, which is provided by a function mapping from a context, for instance, me uttering a sentence to myself. Names, for Kaplan, have a constant referent since they have a fixed character. It has been argued that names with multiple bearers do not have a constant character [53].

Even though the question of whether there are rigid and non-rigid designators in programming languages is an interesting one, the analysed semantics of programming languages do not take into consideration anything similar to contexts of utterance, such as some sort of "context of execution": meaning semantics associated with program constructs are the same, independent of the machine executing the program.

The purpose of the semantics of programming languages is to enable reasoning about correctness at an abstract level. It is true that once a given program is implemented on a physical machine, the chosen semantics (especially operational semantics) can be used to map from program instructions to machine executions. However, identifying the referents (machine executions) is not a job carried out by the semantics themselves; semantics are certainly needed, but something more seems required.

According to [19], models of reference are not to be intended as mutually exclusive, but they can be coupled, when needed, to overcome the inner limitations of each model, depending on the context. For instance, a *causal model* of reference can be associated with Millianism and the view that proper names are some sort of "tags" associated with names. As recalled in the previous section, according to this view, the meaning of a name is just its referent. However, a problem was left open as to who is labelling or tagging objects in the world. According to the causal model of reference, in its more known form due to Kripke [44] but also independently introduced by [55,56], there is an initial "baptism" with which a referent is given a name; the current use of a name is justified since one can trace a causal chain of usage of referents back to the original baptism.

The causal model of reference still sees reference as a one-to-one or a many-to-one correspondence; once a name is associated with a referent, that name is always used to refer to that referent. It follows that this model does not fit as well with the many-

to-many relations characterizing programming language semantics and implementing physical machines.

Finally, an *intentional model* of reference understands the ability of language expressions to be related to the intentions of speakers and their usage of those expressions to refer to intended referents. The intentional model is also known as the *Gricean approach*, given its plain relation with the theory of the meaning of Paul Grice [57], and later systematized in [58]. This model of reference was initially developed for so-called impure indexicals, such as pronouns and demonstratives, for which the context of utterance may not be sufficient to point to the direct referent. By saying 'I', the context of utterance (me saying 'I') determines the referent (myself); by saying 'she' or 'that', more than one referent may fit with the context of utterance. The intentional model proposes to solve this problem by referring to the speaker's mental state, to the listener's understanding of the speaker's mental state, and to the listener's identification of the referent.

Intentions also play a fundamental role in the development and verification of computational systems [2]. *Intention* is understood as the first level of abstraction defining a computational system, and it roughly corresponds to the mental act with which stakeholders conceive and ask for a certain system to be developed [59]. Intentions are articulated and formalised into *program specifications* as a second level of abstraction; specifications are realised into a set of *algorithms*, which are themselves realised into a *high-level language program*. The program is compiled into *machine code*, which is finally implemented and executed by a *computational device*, which is the last level of abstraction. One first consideration is that each of the listed levels of abstraction corresponds to a multiplicity of lower levels. So, for instance, given an algorithm, this can be realised in a multiplicity of different programs using different programming languages. A low-level language program can be executed by different machines. The choice of the level of abstraction that realises a higher one is made on an intentional basis, and it is carried out by the developers. Intentions of stakeholders and developers flow all the way down from intentions to physical computational machines.[19]

Accordingly, an intentional theory of reference can be associated with a form of descriptivism for associating referents with different senses of program instructions. A piece of code is associated by language semantics with states and transitions of an abstract machine; the abstract machine, as shown, may correspond to a multiplicity of physical computational devices, so the meaning (and the different senses associated with it) of the instructions alone does not suffice to identify the referent. It is on an intentional basis that, during program verification, an abstract machine is made to correspond with a physical one.

Software verification cannot be reduced to the algorithmic search for the validity of the specification throughout the model. The reason is that, as specified above, abstract machines, by being abstract, often contain spurious paths, so it may well be the case that a path showing the validity, or the violation, of a specification, does not correspond to any physical machine execution. During verification, software engineers must make sure that those paths are not spurious; this is conducted by executing the program on a machine and observing the executions corresponding to those model paths. For this reason, it has been argued that formal verification always requires empirical testing [61]. It is the developer's intention that allows her to use an abstract machine to refer to a physical one.

5. Concluding Remarks

The first conclusion is that senses alone do not suffice to pick out a referent for programming languages; developers' intentions are needed as well. Strictly speaking, Fregean's slogan that sense determines reference does not hold true for programming

languages. Clearly, this conclusion holds as long as one limits herself in considering the main theories of meaning and of reference as they have been historically discussed in the philosophy of language, as carried out in this paper. The meaning/reference problem involves ongoing debate with new emerging definitions.[20]

A second conclusion is that high-level language programs do refer by means of their language semantics and some intentional act, but *not to an external world*: they can be made to refer to physical machines executing them. One may wonder why the intentional theory of reference cannot be used to map from the program code to objects of the external world, such as mapping a given variable *temp* to an external temperature. The reason is that, in the presented account, the intentional acts of developers are bound by the semantics of the given programming language, which relates programs to the operations of an abstract machine, which, in turn, potentially refers to a multiplicity of physical machines. The intentional act of developers identifies one given machine with the aim of performing verification. This is not to say that computer programs cannot represent external magnitudes, but that if they do, they do not refer to them by virtue of their language semantics! [21] This conclusion is restricted to simulative programs only, which have been the focus of this paper and of the discussion of the program as a theory thesis. A different kind of analysis is deserved for so-called reactive programs, which continuously interact with the environment and whose verification, therefore, involves reference to external magnitudes.

A given simulative program can be used for scientific purposes to refer to or represent an empirical target system, but this is carried out by means of a different theory of reference. The causal model of reference can, for instance, be used to make, say, Newell and Simon's logic theory refer to an agent's mental solution strategy. Moor argued that programs can be epistemologically identified not with scientific theories but with models of empirical systems, provided that mappings between program and empirical systems are established [23]. But, those mappings cannot be established by means of the semantics of the programming language.

In contrast to the dichotomy of the internalist/externalist view of programs, which sees semantic programs as referential, programming languages are semantic but internal, that is, not referring to an external world. It could be objected that machines do operate in the external world and that some of their operations involve the external world. However, in the debate between the internalist and externalist views of programs, the external world is meant as an empirical system different to the machine itself, as it is the human mind in Newell and Symon's view on programs as cognitive theories.

Interestingly, the idea that a language might be semantic but not referential has also been contended for natural languages, for instance, by Chomsky [64]. Chomsky maintains that languages need not be referential in order to have meaning; the meaning of, say, a name is provided without the necessity of identifying a referent in the external world. His argument, in brief, is that meanings are individuated by syntax; syntax and semantics are inseparable or intrinsic to each other.

A similar position has also been held in the context of the philosophy of computing by Piccinini [65]. The author maintains that syntactical rules regulate how to construct strings of symbols from strings of symbols and that those strings do not convey any content. The machine where the strings are stored is able, by executing the program, to generate new strings from the stored ones. The newly generated strings are considered by Piccinini as internal semantics; they are semantics because they assign content to strings in terms of their effects on the machine (the new strings); it is internal because the content is not provided in terms of external objects and the properties thereof but rather in terms of components and processes internal to the machine.

A third and final conclusion is that whether computer programs can be considered as theories does not depend on their language semantics. The latter allows one to consider programs as, at best, theories of operations of implementing machines, a position already maintained by Turner [50]. It remains to be investigated, as a future development, what kind of referential theory, not related to programming language semantics, lets computer programs be considered as theories of empirical systems.

Funding: This research was partially funded by the Prin 2022 project—"SMARTEST—Simulation of Probabilistic Systems for the Age of the Digital Twin" PRIN_20223E8Y4X_002, CUP_J53D23019490006, awarded by the Italian Ministry of University and Research and partially by the Project PROGRAMme ANR-17-CE38-0003-01.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Acknowledgments: I am very grateful to the members of the PROGRAMme project for discussions and feedback on an earlier version of this paper and to the Editors and reviewers for their insightful comments.

Conflicts of Interest: The author declares no conflict of interest.

Notes

1 As extensively shown in the paper, the syntax of a programming language corresponds to the set of rules determining the well-formed statements of the language; semantics amount to providing rules assigning meaning to those statements.

2 This paper recalls the main theories of meaning and of reference, respectively, in Sections 3 and 4. The reader wondering what is meaning and what is reference will find in those sections the different definitions of the two terms that have been provided by the philosophy of language.

3 The constructs of a programming language are to be intended as the building blocks of the language; they include variables, data types, control structures, or functions. This paper mainly considers control structures, particularly the statements of imperative languages.

4 Here, Moor uses the word 'model' in the sense it assumes the philosophy of science as a representational entity referring to a specific empirical system.

5 The philosophy of computer science has delved into the notion of implementation, providing different definitions [2]. In general terms, an implementation is any physical system realising a computational process.

6 In computer science, a specification is a statement, expressed in natural or formal language, declaring the requested behaviours of a computational system to be developed. Once developed, the system is then said to be correct in case it behaves as prescribed by its specifications [24].

7 This is position notoriously supported by James Fetzer [7].

8 See also [3] (ch. 10) for a helpful introduction to operational semantics.

9 Notice that the rule-based form $\frac{premise_1,\ldots,premise_n}{conclusion}$ is used to express the conditional form of semantic rules.

10 It is beyond the scope of this paper to analyse model-checking techniques; for a complete introduction, the reader may refer to [28].

11 The reader should be careful to notice here that the word 'model' is not used in the sense of scientific models, i.e., as a representational entity. Both domains for denotational semantics and abstract machines for operational semantics are mathematical structures set-theoretically conceived.

12 To provide a brief example, consider a program computing the factorial $n!$ of any natural number n; its specification would declare $n \geq 0$ as a pre-condition and $output = n!$ as a post-condition. In this case, the domain is the set of natural numbers and the semantic function is the function $F(n) = n!$. Proving correctness means demonstrating that, for any $n \geq 0$, the function satisfies the post-condition.

13 Returning to the factorial example, given the precondition $n \geq 0$, the specification demands that the postcondition be $output = n!$.

14 It is called Millian–Russellian semantics after John Stuart Mill and his idea that the meaning of a proper name is simply its referent, and after Bertrand Russell who formerly endorsed this view of meaning in *The Principles of Mathematics*.

[15] It is a well-known fact that Frege himself did not provide a precise definition of what senses are. The meaning of a sentence depends on its truth condition, determined by the proposition, or content, it expresses. However, content is not sufficient to determine meaning; one should also consider the way such content is presented, namely its sense.

[16] In other words, the abstract machine is intended to be the meaning (and not the referent) of programming languages, which is presented in different ways (Fregean senses) by different language semantics. Different semantics present, in different ways, the *same* abstract machine assumed here by the definitional priority principle, according to which all semantics must adhere to operational semantics.

[17] There are other theories of meaning which have not been discussed in this section. They include the *character theory* of meaning and the *intensional theory* of meaning. Character theory, by also being a theory of reference, will be discussed in Section 4. The intensional theory of meaning refers to *possible world semantics* [45] that help us understand the meaning of a natural language expression as an *intension*. An intension is a function mapping from possible worlds, i.e., the possible state of things, to referents. The meaning of the expression 'the oldest electronic digital computer' is accordingly a function mapping from the actual world to the Atanasoff–Berry computer. The meaning of such an expression is not directly the referent, namely the ABC computer, since one may well not know what the oldest digital computer is, and nonetheless understand the expression. None of the semantics of programming languages provide functions mapping from the possible state of things to referents; again, what those semantics do is map program constructs to propositional entities.

[18] In any case, by assuming the definitional priority of operational semantics, one may consider the transitions of an abstract machine as the meaning of any programming language.

[19] This aspect has been deeply investigated by Turner [3,24] by arguing that each level of abstraction can be understood as a specification for lower levels in that it defines what lower levels should accomplish. What makes any such a level a specification is what he calls its *intentional stance*, that is, the act with which it is taken to have "correctness jurisdiction" over lower levels of abstraction. One should be careful to distinguish the notion of intentional stance used in the philosophy of computer science from the notion of intentional stance as introduced in the philosophy of the mind by Daniel Dennett [60]. According to the latter, beliefs and desires are ascribed to rational agents in order to be able to predict and explain their behaviours.

[20] An example of a quite different theory of reference is *reference magnetism* [62], originally proposed by David Lewis [63], according to which the metaphysical structure of reality influences the way words refer to objects.

[21] I am very grateful to one reviewer that helped me ro clarify this important point.

References

1. Rapaport, W.J. *Philosophy of Computer Science: An Introduction to the Issues and the Literature*; John Wiley & Sons: Hoboken, NJ, USA, 2023.
2. Angius, N.; Primiero, G.; Turner, R. The Philosophy of Computer Science. In *The Stanford Encyclopedia of Philosophy*; Zalta, E.N., Nodelman, U., Eds.; The Metaphysics Research Lab, Department of Philosophy, Stanford University: Stanford, CA, USA, 2025.
3. Turner, R. *Computational Artifacts. Towards a Philosophy of Computer Science*; Springer: Berlin/Heidelberg, Germany, 2018.
4. Hill, R.K. What an algorithm is. *Philos. Technol.* **2016**, *29*, 35–59.
5. Eden, A.H.; Turner, R. Problems in the ontology of computer programs. *Appl. Ontol.* **2007**, *2*, 13–36.
6. Stephanou, H. About the "Trinity Thesis" Regarding the Ontology of Computer Programs. *Philos. Technol.* **2020**, *33*, 323–330.
7. Fetzer, J.H. Program verification: The very idea. *Commun. ACM* **1988**, *31*, 1048–1063.
8. Angius, N. The problem of justification of empirical hypotheses in software testing. *Philos. Technol.* **2014**, *27*, 423–439.
9. Floridi, L.; Fresco, N.; Primiero, G. On malfunctioning software. *Synthese* **2015**, *192*, 1199–1220.
10. Colburn, T.; Shute, G. Abstraction in computer science. *Minds Mach.* **2007**, *17*, 169–184.
11. Angius, N. Abstraction and idealization in the formal verification of software systems. *Minds Mach.* **2013**, *23*, 211–226.
12. Eden, A.H. Three paradigms of computer science. *Minds Mach.* **2007**, *17*, 135–167.
13. Primiero, G. *On the Foundations of Computing*; Oxford University Press: Oxford, UK, 2019.
14. Gotterbarn, D.; Miller, K.; Rogerson, S. Software engineering code of ethics. *Commun. ACM* **1997**, *40*, 110–118.
15. Koepsell, D.R. *The Ontology of Cyberspace: Philosophy, Law, and the Future of Intellectual Property*; Open Court Publishing: Chicago, IL, USA, 2003.
16. Floridi, L.; Sanders, J.W. On the morality of artificial agents. *Minds Mach.* **2004**, *14*, 349–379.
17. Frege, G. O n Sense and Reference. In *Translations from the Philosophical Writings of Gottlob Frege*; Geach, P., Black, M., Eds.; Basil Blackwell: Oxford, UK, 1960; pp. 56–78.
18. Speaks, J. Theories of Meaning. In *The Stanford Encyclopedia of Philosophy*; Zalta, E.N., Nodelman, U., Eds.; The Metaphysics Research Lab, Department of Philosophy, Stanford University: Stanford, CA, USA, 2024.
19. Michaelson, E. Reference. In *The Stanford Encyclopedia of Philosophy*; Zalta, E.N., Nodelman, U., Eds.;The Metaphysics Research Lab, Department of Philosophy, Stanford University: Stanford, CA, USA, 2024.
20. Newell, A.; Simon, H.A. *Human Problem Solving*; Prentice-Hall: Englewood Cliffs, NJ, USA, 1972.

21. Johnson-Laird, P.N. *The Computer and the Mind: An Introduction to Cognitive Science*; Harvard University Press: Cambridge, MA, USA, 1988.
22. Pylyshyn, Z.W. *Computation and Cognition: Towards a Foundation for Cognitive Science*; MIT Press: Cambridge, MA, USA, 1984.
23. Moor, J.H. Three myths of computer science. *Br. J. Philos. Sci.* **1978**, *29*, 213–222.
24. Turner, R. Specification. *Minds Mach.* **2011**, *21*, 135–152.
25. Szabó, Z.G.; Thomason, R.H. *Philosophy of Language*; Cambridge University Press: Cambridge, UK, 2018.
26. Fernández, M. *Programming Languages and Operational Semantics*; King's College Publications: London, UK, 2004.
27. Plotkin, G.D. *A Structural Approach to Operational Semantic*; Aarhus University: Aarhus, Denmark, 1981.
28. Baier, C.; Katoen, J.P. *Principles of Model Checking*; MIT Press: Cambridge, MA, USA, 2008.
29. Scott, D. *Outline of a Mathematical Theory of Computation*; Oxford University Computing Laboratory, Programming Research Group: Oxford, UK, 1970; pp. 169–176.
30. Scott, D.S.; Strachey, C. *Toward a Mathematical Semantics for Computer Languages*; Oxford University Computing Laboratory, Programming Research Group: Oxford, UK, 1971; Volume 1.
31. Stoy, J. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*; MIT Press: Cambridge, MA, USA, 1977.
32. Hoare, C.A.R. An axiomatic basis for computer programming. *Commun. ACM* **1969**, *12*, 576–580.
33. Milner, R. Processes: A Mathematical Model of Computing Agents. In *Logic Colloquium '73 (Studies in the Logic and the Foundations of Mathematics, 80)*; Rose, H.E., Shepherdson, J.C., Eds.; North-Holland: Amsterdam, The Netherlands, 1975; pp. 157–174.
34. Abramsky, S. Semantics of Interaction: An Introduction to Game Semantics. In *Proceedings of the 1996 CLiCS Summer School*; Dybjer, P., Pitts, A., Eds.; Isaac Newton Institute/Cambridge University Press: Cambridge, UK, 1997; pp. 1–31.
35. Hyland, J.M.E. Game Semantics. In *Proceedings of the 1996 CLiCS Summer School*; Dybjer, P., Pitts, A., Eds.; Isaac Newton Institute/Cambridge University Press: Cambridge, UK, 1997; pp. 131–184.
36. Cardone, F. Games, Full Abstraction and Full Completeness. In *Stanford Encyclopedia of Phiolosophy*; Zalta, E., Ed.; The Metaphysics Research Lab, Department of Philosophy, Stanford University: Stanford, CA, USA, 2021.
37. Hintikka, J.; Sandu, G. Game-theoretical semantics. In *Handbook of Logic and Language*; North-Holland: Amsterdam, The Netherlands, 1997; pp. 361–410.
38. Hintikka, J. *The Game of Language: Studies in Game-Theoretical Semantics and Its Applications*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2012; Volume 22.
39. Winskel, G. *The Formal Semantics of Programming Languages: An Introduction*; MIT Press: Cambridge, MA, USA, 1993.
40. Frege, G. *Begriffsschrift*; Beaney, M., Ed.; *The Frege Reader*; Basil Blackwell: Oxford, UK, 1997; pp. 47–79.
41. Hodges, W. *A Shorter Model Theory*; Cambridge University Press: Cambridge, UK, 1997.
42. Tarski, A. Contributions to the theory of models, I. *Indag. Math.* **1954**, *16*, 572–581.
43. Davidson, D. *Inquiries into Truth and Interpretation*; Oxford University Press: Oxford, UK, 1984.
44. Kripke, S. *Naming and Necessity*; MIT Press: Cambridge, MA, USA, 1972.
45. Lewis, D. General Semantics. *Synthese* **1970**, *22*, 18–67.
46. Russell, B. On Denoting. *Mind* **1905**, *14*, 479–493.
47. Kaplan, D. Words. *Proc. Aristot. Soc.* **1990**, *64*, 93–119.
48. Strawson, P. On Referring. *Mind* **1950**, *59*, 320–344.
49. Donnellan, K. Reference and Definite Descriptions. *Philos. Rev.* **1966**, *75*, 281–304.
50. Turner, R. Programming languages as mathematical theories. In *Thinking Machines and the Philosophy of Computer Science: Concepts and Principles*; Vallverdú, J., Ed.; IGI Global: New York, NY, USA, 2010; pp. 66–82.
51. Clarke, E.M.; Grumberg, O.; Peled, D.A. *Model Checking*; The MIT Press: Cambridge, MA, USA, 1999.
52. Clarke, E.; Grumberg, O.; Jha, S.; Lu, Y.; Veith, H. Counterexample-guided abstraction refinement. In *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, 15–19 July 2000*; Proceedings 12; Springer: Berlin/Heidelberg, Germany, 2000; pp. 154–169.
53. Pelczar, M.; Rainsbury, J. The Indexical Character of Names. *Synthese* **1998**, *114*, 293–317.
54. Kaplan, D. Demonstratives. In *Themes from Kaplan*; Almog, J., Perry, J., Wettstein, H., Eds.; Oxford University Press: New York, NY, USA, 1989; pp. 481–563.
55. Geach, P. The Perils of Pauline. *Rev. Metaphys.* **1969**, *23*, 287–300.
56. Donnellan, K. Proper Names and Identifying Descriptions. *Synthese* **1970**, *21*, 335–358.
57. Grice, P. Meaning. *Philos. Rev.* **1957**, *66*, 377–388.
58. Stine, G. Meaning Other Than What We Say and Referring. *Philos. Stud.* **1978**, *33*, 319–337.
59. Primiero, G. Information in the philosophy of computer science. In *The Routledge Handbook of Philosophy of Information*; Floridi, L., Ed.; Routledge: London, UK, 2016; pp. 90–106.
60. Dennett, D.C. *The Intentional Stance*; MIT Press: Cambridge, MA, USA, 1989.

61. Angius, N. On the mutual dependence between formal methods and empirical testing in program verification. *Philos. Technol.* **2020**, *33*, 349–355.
62. Azzouni, J. Reference Magnetism. In *The Rule-Following Paradox and Its Implications for Metaphysics*; Springer: Cham, Switzerland, 2017; pp. 55–72.
63. Lewis, D. Putnam's paradox. *Australas. J. Philos.* **1984**, *62*, 221–236.
64. Chomsky, N. *New Horizons in the Study of Language and Mind*; Cambridge University Press: Cambridge, UK, 2000.
65. Piccinini, G. Computation without representation. *Philos. Stud.* **2006**, *137*, 204–241.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.