Global Philosophy (2023) 33:13
https://doi.org/10.1007/s10516-023-09662-6

ORIGINAL PAPER

On Giving Meanings to Programs

Felice Cardone[1]

Received: 30 September 2022 / Accepted: 22 November 2022

Abstract
In a short section on the semantics of programs within his discussion of program correctness, Primiero seems to endorse the received view on the Scott-Strachey approach to denotational semantics as directly related to correctness. While this is true to some extent, I argue that the mathematical entities associated with programs play a lesser role in reasoning on program correctness, while the mathematical foundations of denotational semantics, namely the theory of domains, have contributed significantly to the conceptual understanding of programming and of computation in general

Keywords Foundations of computing · Denotational semantics · Scott domains

1 Introduction

I would like to focus on a tiny part of Primiero's monograph, namely his mention of denotational semantics in Chapter 7. This is due in part to autobiographical reasons, as I have been working for a few decades along the lines of the denotational approach, mostly in its applications to type theories for programming languages. Leaving aside this sentimental motivation, I would like to comment extensively on a statement (Primiero 2019, p. 96) which summarizes the received view of the Scott-Strachey contribution and starts a short section on semantics as one of several approaches to correctness:

> To establish a precise correspondence between programs and mathematical entities entirely independently of implementation was also the task of the denotational semantics developed by Scott and Strachey in the 1960s.

My purpose in this paper will be to argue that the mathematical entities associated with programs play a lesser role as practical tools for reasoning on program correctness, while the mathematical foundations of denotational semantics, namely the theory of

✉ Felice Cardone
felice.cardone@unito.it

[1] Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy

Published online: 31 January 2023

domains, have contributed significantly to the conceptual understanding of programming and of computation in general. This will also give us the occasion to touch on several foundational issues that recur throughout Primiero's book, such as implementation, miscomputation and correctness.

2 The Semantical Problem for Programming Languages

2.1 A Simple Language and its Denotational Semantics

In order to give an idea of the denotational approach, I start from a miniature language whose purpose is not only to give an idea of the general method, but also to expose some of what I regard as foundational weaknesses in that approach.

The example is suggested by §33 of Wittgenstein's Brown Book (1936), and consists of basic instructions for moving one step in each direction N, S, E and W along a grid with integer coordinates. The basic instructions can then be composed to describe paths: we can imagine this as a toy instruction language for a robot, or a fragment of a language for choreography; a path is then an *algorithm* for reaching the end point from the starting point.

It is easy to interpret these instructions operationally, each as the performance of the corresponding move. The denotational semantics of an instruction is defined as a function that maps the starting point of the corresponding path to its end point:

$$\begin{aligned}
[\![\uparrow]\!](x, y) &= (x, y + 1),\\
[\![\downarrow]\!](x, y) &= (x, y - 1),\\
[\![\rightarrow]\!](x, y) &= (x + 1, y),\\
[\![\leftarrow]\!](x, y) &= (x - 1, y).
\end{aligned}$$

The interpretation is extended to all finite instruction sequences by structural recursion:

$$[\![S_1;S_2]\!](x, y) = [\![S_2]\!](x', y'), \text{where } (x', y') = [\![S_1]\!](x, y).$$

Observe that this extension makes the interpretation, literally, *compositional*: the meaning of a compound instruction is computed as a function of the meanings of its components. For example, we have

$$[\![\uparrow ; \rightarrow ; \downarrow]\!](x, y) = (x + 1, y)$$

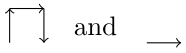
as a consequence of the following steps:

$$\begin{aligned}[[\uparrow ; \rightarrow ; \downarrow]](x, y) = & \\ & = [[\rightarrow ; \downarrow]][[\uparrow]](x, y) \\ & = [[\rightarrow ; \downarrow]](x, y + 1) \\ & = [[\downarrow]][[\rightarrow]](x, y + 1) \\ & = [[\downarrow]](x + 1, y + 1) \\ & = (x + 1, y)\end{aligned}$$

These calculations show the equivalence of the compound instruction $\uparrow ; \rightarrow ; \downarrow$ with the basic instruction $\rightarrow$: indeed, for all (x, y), we have

$$[[\uparrow ; \rightarrow ; \downarrow]](x, y) = (x + 1, y) = [[\rightarrow]](x, y)$$

Observe that this equivalence follows from the decision to regard instructions, extensionally, as transformations from initial to final points. The intepretation of instructions as whole paths is closer to an operational, intensional viewpoint, and makes the equivalence above fail, because the paths



have the same endpoints but consist of disjoint moves.

Our simple language exhibits several familiar features of programming languages: it consists of *instructions* that are interpreted as *state transformations* (here states are points on the cartesian plane with integer coordinates); instructions can be *composed* and the interpretation of instructions is compositional; interpretation automatically defines semantical equivalence of instructions. Furthermore, there is a constrast between paths as intensional meanings of instructions and their extensional traces consisting of the coordinates of their extremities. My claim is that, despite its simplicity, the contemplation of this language exposes general foundational problems of the denotational approach.

2.2 Meaning, Correctness and Implementation

A first question concerning this toy language is whether the denotational *interpretation* that I have sketched is helpful in defining an *implementation* of the language.[1] Observe that implementation here may consist in several different things, according to the level of abstraction. It might consists in a human performing the instructions according to their interpretations, or in programming a robot to obey the instructions, or in a robot obeying the instructions. In any case, implementation has to pass through a semantic interpretation specifying the meaning of instructions.

[1] There is a considerable amount of literature on this topic in the philosophy of computing, summarized and discussed in (Primiero 2019, Ch. 11).

Now, one basic observation is that these meanings have been specified by means of transformations of pairs of integer numbers, but these pairs in themselves do not have any geometric significance. Their geometric rendering *presupposes* that we already know what it means, for example, to move one step right, i.e., assumes that the meaning of the instruction → is already understood, hence that it is more basic, for the clause $[\![\rightarrow]\!](x, y) = (x + 1, y)$ to make any sense. Indeed, how could we teach this language by explaining the meaning of the instructions by means of this denotational interpretation?

The same applies to the interpretation of ordinary programming constructs. Consider for example the `while` instruction:

```
while (b) do S
```

where `b` is a boolean expression and `S` an instruction. Intuitively, this instruction is executed by performing the following actions starting form a *state* σ:

1. evaluate `b` in σ;
2. if the result is `true`, then execute `S` from σ and go to 1; otherwise stop.

Denotationally, we must interpret this instruction as a state transformation $\phi = [\![\texttt{while (b) do S}]\!]$ that satisfies the fixed point equation

$$\phi = \lambda\sigma.\ \text{if } [\![\texttt{b}]\!]\sigma \text{ then } \phi([\![\texttt{S}]\!]\sigma) \text{ else } \sigma$$

This fixed point is the mathematical representation of the process of repeatedly performing ϕ until `b` becomes false. Therefore this example is subject to the same circularity as before: understanding the interpretation of the `while` instruction amounts to having grasped the technique of iterating a transformation, which is the intuitive meaning of this instruction.

Consider Wittgenstein's criticism of Russell's attempt to reduce the practice of decimal arithmetic to its unary encoding (Wittgenstein 1956, §II 3): I believe that the same arguments can be turned against any attempt to reduce the meaning of programming constructs to a denotational specification of their interpretation, if meaning has to be what allows us to teach a language to people who haven't seen it before. Instead, it is most likely that teaching such a basic language as that in the example can be achieved by developing a practice that consists in making appropriate *gestures*.[2]

The original motivation for the Scott-Strachey approach included using denotational meanings as a standard against which to assess correctness of implementation, for example of a compiler. This is what normally is taken to be the *normative*

[2] Gestures have increasingly been recognized as playing a fundamental cognitive role in the constitution of mathematical objects: I expect that this applies even more appropriately to computing. On this, see (Longo 2005) and the references cited therein.

role of semantics, the yardstick allowing to recognize those cases where some kind of miscomputation must have taken place, originating for example from systematic misunderstandings between language designers and implementers. This normative value fails in two extreme cases of denotational specifications. First, as I have tried to argue above, for very basic instructions like those of the wittgensteinian example, because it is the intuitive meaning of an instruction like → that allows to understand the denotational interpretation of →, and not the other way around. Second, for real programming languages with, say, a rich type discipline, higher-order functions, modules and other common paraphernalia, because the semantic clauses involved in the interpretation of their constructs lack the perspicuity that is needed to guarantee the stability of meanings by applying the recursive clauses of the semantic definition. In these cases very early we need to rely upon programmed tools that help in calculations, and from an epistemological standpoint this raises the same debates as the use of computer in mathematical proofs like that of the four color theorem, see (Tymoczko 1979) for an early discussion.

That denotational semantics cannot be the target of a reductive explanation of meaning for programs was, in a sense, clear from the start. A possible way out is to regard denotational specifications as means of associating mathematical objects to programming constructs in such a way as to be able to formally reason about the latter, for example the denotational meaning of the `while` instruction as a (least) fixed point makes possible to prove its properties by means of fixed-point induction and related techniques (Manna and Vuillemin 1972).

But here a new problem arises, namely that of guaranteeing that the denotational values of program phrases coincide with what is to be expected from executing them, their operational semantics. While the original project of denotational semantics eschewed any reference to implementation of programs, ultimately it is with the outcome of running a program on a machine that its denotational value must conform. Of course, not a physical machine but a mathematical model of it, an alternative source of normativity.

2.3 Operational and Denotational Semantics

In the field of programming semantics, the coexistence of the denotational and operational approaches has been granted since the dawn of the subject (Scott 1970b):

> the operational aspects cannot be completely ignored. The reason is obvious: in the end the program still must be run on a machine – a machine which does not possess the benefit of "abstract" human understanding, a machine that must operate with finite configurations.

What is needed is a formalism for operational semantics that is abstract enough as to hide irrelevant details of the underlying machine, and this has eventually been found in the structural operational semantics described in (Plotkin 2004b).[3] Here

[3] See (Plotkin 2004a) for a historical introduction.

the interpretation is expressed by means of rules allowing to infer judgements of the form

$$(\mathtt{S}, \sigma) \Downarrow \sigma' \tag{1}$$

where $\mathtt{S}$ is an instruction and σ, σ' are states, or of the form

$$(\mathtt{e}, \sigma) \Downarrow v \tag{2}$$

where $\mathtt{e}$ is an expression (in particular, a boolean expression) and v is a value (in particular one of the boolean values `true` or `false`). Then we can specify as follows the interpretation of the `while` instruction:

$$\frac{(\mathtt{b}, \sigma) \Downarrow \mathtt{true} \quad (\mathtt{S}, \sigma) \Downarrow \sigma' \quad (\mathtt{while\ (b)\ do\ S}, \sigma') \Downarrow \sigma''}{(\mathtt{while\ (b)\ do\ S}, \sigma) \Downarrow \sigma''} \qquad \frac{(\mathtt{b}, \sigma) \Downarrow \mathtt{false}}{(\mathtt{while\ (b)\ do\ S}, \sigma) \Downarrow \sigma}$$

Clearly these rules capture the intended behavior of the `while` instruction and minimize the reference to abstract mathematical entities, although there is an implicit assumption of a mathematical nature in the background, namely that rules are interpreted as giving an *inductive* definition of the true judgements, which corresponds to taking, also in this case, the least fixed point of a monotonic set-theoretic operator.

The required harmony of the denotational and the operational semantic specifications can now be made formal, by considering a fundamental relation between the two: clearly we must have for any σ that

$$(\mathtt{S}, \sigma) \Downarrow \sigma' \quad \Rightarrow \quad [\![\mathtt{S}]\!]\sigma = \sigma'$$

The converse property is *adequacy*: whenever the denotational semantics of an instruction gives a value, it must be possible to obtain that same value by applying the operational rules:

$$[\![\mathtt{S}]\!]\sigma = \sigma' \quad \Rightarrow \quad (\mathtt{S}, \sigma) \Downarrow \sigma'$$

The strictest property is *full abstraction*, which states the coincidence of two equivalence relations on instructions induced by the denotational and operational semantics, respectively. The first relation is easily defined, by setting $\mathtt{S}_1 \approx_{\mathrm{den}} \mathtt{S}_2$ if and only if $[\![\mathtt{S}_1]\!]\sigma = [\![\mathtt{S}_2]\!]\sigma$ for all states σ. Operational equivalence between instructions $\mathtt{S}_1$ and $\mathtt{S}_2$, written $\mathtt{S}_1 \approx_{\mathrm{op}} \mathtt{S}_2$ holds precisely when, for all contexts $C[\ \ ]$ in which we can plug these instructions getting a *program* we have

$$C[\mathtt{S}_1] \Downarrow \sigma \quad \Leftrightarrow \quad C[\mathtt{S}_2] \Downarrow \sigma$$

where the assumption that $C[\ \ ]$ yields a program allows us to omit the state parameter (think of a closed formula in first-order logic).[4]

[4] There is a huge literature on full abstraction, follow the references in (Cardone 2021).

A fully abstract denotational semantics would be completely justified operationally, and would allow to use the mathematical techniques made available by domain theory to prove operational properties. It turns out that full abstraction fails for the natural denotational interpretation of a very general model of (functional) programming language closely related to the typed lambda-calculus proposed by Scott as a formalism for computable functions (Scott 1969b, a), which is the subject of the two fundamental studies (Milner 1977; Plotkin 1977). The attempts at solving the problem have stimulated efforts to understand the foundations of denotational semantics and the intuitive justification of its formal tools, which I take to be its most significant contribution.[5]

3 The Foundations of Denotational Semantics

There were two themes that merged in Scott's project of denotational semantics. On the one hand, there was the aim of providing a mathematical basis to the calculations needed in the compositional interpretation of programming language constructs defined in (Strachey 2000; Scott and Strachey 1971). This included the equations between semantic domains required by the interpretation of higher-order procedures, in particular the equation $D = [D \to D]$, where $[D \to D]$ is a suitable domain of endofunctions of D. The solution to this equation is needed for the interpretation of the type-free lambda-calculus into which, as the work of Peter Landin (1965) had shown in the meantime, can be encoded most constructs of languages such as Algol 60. This equation allows to consider every element $d \in D$ as a function from D to itself, making possible the application $d(d)$ of d to itself. Scott's solution was a very nice and successful piece of conceptual analysis whose steps are worth summarizing and comparing with the contrasting view of Haskell B. Curry concerning the interpretation of type-free theories of functions like the lambda-calculus or the theory of combinators. This is the second theme in denotational semantics, namely the foundational concerns about the interpretation of such type-free formalisms by means of ordinary mathematics: it is well-known that Scott regarded them as mathematically flawn, and indeed devoted a series of papers to a reconstruction of higher-order computability by means of typed lambda-calculi (Scott 1969b, a). Ultimately, the contrast between the ideas of Scott and Curry lies in a basic philosophical disagreement on giving meaning to formal notions.

3.1 Curry on Formal Systems

Curry developed throughout his lifetime a *formalist* approach to the philosophy of mathematics (Curry 1951) based on a notion of *formal system* that is the object of

[5] The difficulty of building fully abstract models of programming languages has suggested an alternative approach that tries to carve approximation (pre)order and (some) limits from operational semantics and reformulate from these the proof principles that hold for domains (Pitts 1997).

many of his publications, summarized in (Curry 1963, §3C).[6] Some of the remarks that Curry made in relation to formal systems are relevant to semantical issues. First of all, it should be kept in mind that, according to Curry, a formal system is an activity carried out in the language in use, what Curry calls the *U language*, by extending it with nouns and operators to designate the objects of the formal system (its *obs*). From nouns and verbs one can build *sentences* that designate *statements* to the effect that the *predicate* designated by a verb applies to a sequence of obs designated by nouns. The *theorems* are statements obtained by means of rules allowing to infer one conclusion from a (possibly empty) set of premises. This duplication of terminology allows to distinguish grammatical notions of that part of the U language used to communicate the formal system and their abstract correlates. According to Curry, they are not linguistic entities (the U language is not a metalanguage) and no concrete representation is given to them: formal systems are *abstract*. As an example, a formal system for the natural numbers may include an atom 0, an operation S and elementary statements $X = Y$ where X and Y are formal objects. Here '0' is a noun in the U language, the language that I am using throughout the present paper, and 'S' is a unary operator, whereas '=' is a verb. '$0 = 0$' is a sentence designating the statement that 0 equals itself. The important thing to notice is that no other language is involved in this communication beyond a suitable extension of the U language.[7] The linguistic presentation of an abstract formal system is akin to that used for algebraic structures, and indeed Curry's formalism has been regarded as a form of structuralism (Seldin 2011). An *intepretation* of a formal system associates to its statements *contensive* propositions, those "which we understand independently of the formal system in terms of our prior, or at least extrinsic, experience" (Curry 1941, p. 357). The truth criterion for the statements is their derivability according to the rules, which therefore implicitly define the formal meaning of the obs and of the statements of the system. When both can be intepreted so that the system turns out to be applicable to some contensive domain, the system is *acceptable* (Curry 1953).

> The acceptability of a theory is an empirical matter, i.e. we can never be certain of the acceptability or even the contensive validity of a theory for a purpose related to experience [...] We can only entertain the acceptability of a theory as a hypothesis until the discovery of new facts shows that it is untenable.

Speaking years later about combinatory logic and lambda-calculus (Curry 1980):

> While it is true that concocting formalisms entirely without regard to interpretation is probably fruitless, yet it is not necessary that there be "conceptualization" in terms of current mathematical intuitions [...] Combinatory logic [...] did have an interpretation by which it was motivated. The formation of functions from other functions by substitution does form a structure, and this structure it analyzed and formalized.

[6] See also (Meyer 1987) for a review of Curry's philosophy of formal systems.

[7] This is explained very clearly in §IV of Meyer's paper cited above. Curry was always careful in distinguishing between use and mention, as far as to include in his (1963) logic textbook a series of exercises on the proper use of quotation marks.

We may apply this approach to formal systems to the semantics of a programming language. In this case we only need a formal system specifying how its phrases are evaluated, or executed, in order to have a standard against which acceptability can be assessed. There is no need of a further semantical layer assigning formal meanings to program phrases beyond the formal truth of statements of operational semantics of the forms (1) and (2), in particular there is no need of passing through statements of the form $[\![\mathtt{S}]\!]\sigma = [\![v]\!]$, that makes also necessary to interpret contensively the formal objects $[\![\mathtt{S}]\!]\sigma$. The relation of operational semantics to physical implementation of the language is then the same as that between a formal system and the contensive domain which is the target of an interpretation: acceptability takes into account the possibility of miscomputations due to physical failures and makes the application of a mathematical framework a matter of use and practice.

3.2 Scott and Set-Theoretic Semantics

Scott's analysis sets up a framework, based on an abstract notion of *information* content of *partial*, *computable* elements of domains, that allows eventually to recover the syntactic primitives of function calculi (like abstraction and application) as invariants arising in a natural way from the structure of the category of domains. The latter can also be regarded as a type theory[8] including solutions of equations like $D = [D \to D]$. The set-theoretic language in which Scott's axioms for domains are formulated guarantees that the entire framework complies with the "canons of type theory", and "can (and indeed must be) read as a fragment of set theory so that its theorems can be recognized as *valid*" (Scott 1969b, p. 414). Set theory is not a source of meaning in itself, but only in as far as it allows a direct formulation of a framework that, as a whole, is algebraic and can be instantiated in many ways. We shall now see how different instantiations have been helpful in clarifying issues directly relevant to the interpretation of computational phenomena.

The original axioms for domains in (Scott 1970a) were suggested by a picture of computation as increasing information about *partial* elements and of computable functions as monotonically preserving the information. This idea of partiality is related to that of a notion of *approximation* between elements, which can be naturally expressed as a partial ordering:

> [...] we must ask: what exactly is a *data type*? [...] Suppose $x, y \in D$ are two elements of the data type, then the idea is not immediately to think of them as being completely *separate* entities just because they may be *different*. Instead y, say, may be a *better* version of what x is trying to approximate. In fact, let us write the relationship
>
> $$x \sqsubseteq y$$
>
> to mean intuitively that y is *consistent with* x and is (possibly) *more accurate than* x. [...] thus $x \sqsubseteq y$ means that x and y want to approximate the same entity,

[8] An "algebra of types" according to (Scott 1969b, p. 412).

but y gives *more* information about it. This means we have to allow "incomplete" entities, like x, containing only "partial" information (Scott 1970b).

There is an element $\perp$ that does not contribute any information, and the processes that increase information may be infinite but all chains in the approximation ordering are always assumed to converge to an element that cumulates all the information contained in the elements of the chain. Domain theory is thus a theory of approximation and limit for a qualitative notion of information.

4 The Meaning of Meaning

How real is this relation of approximation defined on data types? In other words, is there any operational justification for such a relation? One year before Scott found his models for the lambda-calculus, James H. Morris completed his MIT thesis on "Lambda-calculus models of programming languages" (Morris 1968), where he reformulated for lambda-terms the natural extension relation for partial functions: B extends A (written $A \preceq B$) if, for all contexts $C[\ \ ]$ such that $C[A]$ reduces to a normal form, then also $C[B]$ reduces to the same normal form. This is a preorder on lambda-terms which is preserved by contexts and induces a congruence $\simeq$ in the usual way: while $A \preceq B$ means that in any context B contributes to the computation of a value as much as A and possibly more, $A \simeq B$ means that the two terms are indistinguishable in as far as we are "ultimately interested only in normal forms" as results (Morris 1968, p. 52).

The preorder can be generalized to any setting where there is a notion of (possibly non-terminating) *reduction* relation over expressions, a well-defined notion of *context* that preserves reduction, and a notion of an expression being a *value*, at which reduction stops. Take for example a data type of natural numbers, containing the denotations of numerical expressions. Operationally, there are at least the following choices, depending on what is regarded as a value:

- The "flat" natural numbers: here the reduction of an expression can proceed indefinitely without ever reaching a value, and expressions whose reductions have completely defined values of the form $n = S(S(S(\ldots(0)\ldots)))$. In the associated data type there is an element $\perp$ that corresponds to expressions of the former kind, that do not contribute any information, and the completely defined numbers n having maximal information content; the only possible relation of approximation makes $\perp \sqsubseteq n$.
- The "oblique" natural numbers: here reductions stop at numerical expressions of the form $S(e)$. The definite information provided by the value of an expression e can be of the following types:

  1. e has no value, so it is mapped to $\perp$;
  2. e has value 0, and it is mapped to the corresponding element of the data type;
  3. e has a value of the form $S(e')$: in this case it is mapped to $S(d)$, where d is the value of e'.

In this case there are elements with maximal, therefore incomparable, information content, $0, S(0), S(S(0)), \ldots$, but approximation is finer: we have $\bot \sqsubseteq 0$ and $\bot \sqsubseteq S(x)$ for any x, and $S(x) \sqsubseteq S(y)$ whenever $x \sqsubseteq y$.

This operational analysis of the structure of two possible data types for natural numbers is not the only way of extracting order-theoretic properties of approximation. The general technique employed in these investigations consists basically in the following steps:

- give idealized descriptions of concrete situations in which computationally interesting dynamics involving information can be identified;
- associate to each of these situations a domain describing an approximation structure that captures the appropriate patterns of information dynamics;
- prove a representation theorem stating that there is a perfect match between the structure of idealized situations and that of the appropriate class of domains.

We can instantiate this technique by considering an idealized situation where the elements of the two data types of natural numbers are given by propositional information.

- Propositions for the data type of flat natural numbers have the form $= \mathtt{n}$ for all natural numbers n, where $= \mathtt{n}$ and $= \mathtt{m}$ are inconsistent if $n \neq m$. An element of the data type is given by a consistent set of propositions, therefore it can either be $\bot = \emptyset$ or a singleton $n = \{= \mathtt{n}\}$.
- Propositions for the data type of oblique natural numbers have the forms $\geqslant \mathtt{n}$ and $= \mathtt{n}$. We still have that $= \mathtt{n}$ and $= \mathtt{m}$ are inconsistent if $n \neq m$, but now we have a non-trivial entailment relation between propositions whereby $= \mathtt{n}$ entails $\geqslant \mathtt{k}$ and $\geqslant \mathtt{n}$ entails $\geqslant \mathtt{k}$ if $n \geq k$. Consistent sets of propositions remain consistent if we add propositions entailed by their elements; we can identify the elements of this data type with consistent sets of propositions, that will have one of the following forms:

  1. $\bot = \emptyset$,
  2. $S^n(\bot) = \{\geqslant \mathtt{k} \mid n \geq k\}$ and
  3. $S^n(0) = \{= \mathtt{n}\} \cup \{\geqslant \mathtt{k} \mid n > k\}$ for $n \in \mathbb{N}$.

  Approximation on the data type represented this way is then given by inclusion of sets of propositions. Observe that there is a natural notion of limit of the sequence of elements of the form $S^n(\bot)$ given by the set $\infty = \{\geqslant \mathtt{k} \mid k \in \mathbb{N}\}$.

Such concrete representation of the information content of natural numbers as elements of one of these data types developed into a comprehensive conception of domains as *information systems* in Scott (1982).

The abstract notion of information and the order-theoretic language used by Scott in axiomatizing domains encourage the search for categories of domains

and suitable morphisms motivated by the analysis of concrete situations of information increase for very general notions of computation.[9]

As one more example, I consider a third way of looking at the data type of oblique natural numbers as portraying the information content of a different computational situation. Consider the string rewriting rule

$$N \to 0 \mid SN$$

where we allow infinite derivations of the form $X_0 \to X_1 \to X_2 \to \ldots$ We can associate information increase to derivations, and take derivations themselves as elements of the data type. Approximation can be defined by setting $\zeta \sqsubseteq \xi$ when ζ as a sequence of steps is a prefix of the sequence of steps performed by ξ, with $\perp$ the empty derivation ε. Of course all the derivations having the form $N \to^* S^n 0$ will be incomparable and maximal under this ordering, but there is an infinite increasing chain of derivations of the forms:

$$\varepsilon \sqsubseteq N \to SN \sqsubseteq N \to^2 SSN \sqsubseteq N \to^3 SSSN \sqsubseteq \ldots N \to^\omega SSS\ldots$$

In this example derivations correspond uniquely to the strings that are their end points. It is easy to build situations where there are different derivations leading to the same string that may, however, be regarded as equivalent. Add, for example, the following productions to the above grammar:

$$P \to LR$$
$$L \to N$$
$$R \to N$$

Then rewritings can be performed in parallel when they happen in two different *places*, here L and R. Therefore the two derivations

$$P \to LR \to NR \to 0R \to 0N \to 00 \quad \text{and} \quad P \to LR \to LN \to L0 \to N0 \to 00$$

involve the same *events* of rewriting, only in a different order, and can therefore be regarded as equivalent. The resulting *permutation* equivalence is very general and has important applications in formal language theory (Griffiths 1968), category theory (Hotz 1966; Benson 1975), reduction theory for the lambda-calculus (Lévy 1978) and term-rewriting systems (Huet and Lévy 1979) and also in the theory of concurrent computation (Stark 1989). Even more general than this representation in terms of derivations we can regard the elements of domains as sets of non conflicting events, like in (Nielsen et al. 1981), where

[9] As far as I know, the need of distilling the axioms for categories of domains from the analysis of "actual computational processes" was first pointed out by John Reynolds (1975), who considered nonterminating processes that communicate through channels via sets of *messages*: we take as given a universe of *models* and a relation of *satisfaction* between messages and models, so that the reception of a new message reduces the set of models that satisfy all the messages received so far along the same channel: this gives a representation of the original proposal by Scott (1972) of viewing domains as continuous lattices.

> an event is imagined to occur at a fixed point in space and time; conflict between events is localised in that two conflicting events are enabled at the same time and are competing for the same point in space and time

which allows to apply domain structures to another very general approach to computation, namely Petri nets, whose foundational relevance has not yet been explored in sufficient detail.[10] For example, we have the event structure of "vertical" natural numbers, which correspond to ticks of a clock, whereas approximation corresponds to temporal order,

$$\bot = \text{no tick} \sqsubseteq \text{one tick} \sqsubseteq \text{two ticks} \sqsubseteq \ldots$$

5 Conclusions

I have tried to provide evidence to my claim that the developments outlined above, mainly by means of examples, have been the more significant contribution of denotational semantics to the understanding of computational phenomena, in the form of a mathematical theory of computation as foreshadowed in Scott (1970b). I have contrasted this with the received view of denotational semantics as a tool for program correctness, which was the original way of using it especially in the area of language definition, where this approach had the advantage of showing that the semantics of a programming language and its implementation are different things. I have also tried to suggest that reflection on the conceptual basis of the approach of denotational semantics raises issues of interest for the foundations of computing, proposing implicitly that this conceptual basis should be explored further, with the hope that this exploration may contribute to some of the debates analyzed by Primiero (2019), especially in §7.3 and §11.1.

As a final remark, consider the following quote from (Tarjan 1983, §1.2):

> In order to study the efficiency of algorithms, we need a model of computation. One possibility is to develop a denotational definition of complexity, as has been done for program semantics [...], but since this is a current research topic we shall proceed in the usual way and define complexity operationally.

Recently, some of the investigations in denotational semantics have crossed the bounds of classical domain theory with the invention of game semantics (Abramsky et al. 2000; Hyland and Ong 2000), motivated by the increasing role played by interactivity in computation. This, together with a set of results on the representability of *algorithms* in (theoretical) programming languages, along the lines of (Colson 1991), leads to the hope that denotational methods could also be applied to the investigation of intensional properties of programs (like complexity) and thus provide a

[10] The work of Carl A. Petri and some of his coworkers dealt at a very early stage with the physics of computation, reversibility, resource consciousness, anticipating themes that are a focus of current theoretical research.

different access to other foundational notions that one would like to study "entirely independently of implementation", in the same spirit as denotational semantics.

Funding Open access funding provided by Università degli Studi di Torino within the CRUI-CARE Agreement.

References

Abramsky S, Jagadeesan R, Malacaria P (2000) Full abstraction for PCF. Inf Comput 163:409–470
Benson DB (1975) The basic algebraic structures in categories of derivations. Inf Control 28(1):1–29
Cardone F (2021) Games, full abstraction and full completeness. In: Zalta EN (ed) The stanford encyclopedia of philosophy, 2021st edn. Stanford University, Stanford
Colson L (1991) About primitive recursive algorithms. Theor Comput Sci 83(1):57–69
Curry HB (1941) Some aspects of the problem of mathematical rigor. Bull Am Math Soc 47:221–241
Curry HB (1951) Outlines of a formalist philosophy of mathematics. North-Holland Co., Amsterdam
Curry HB (1953) Theory and experience. Dialectica 7(2):176–178
Curry HB (1963) Foundations of mathematical logic. McGraw-Hill, New York
Curry HB (1980) Some philosophical aspects of combinatory logic. In: Barwise J, Keisler HJ, Kunen K (eds) The kleene symposium. North-Holland Co., Amsterdam, pp 85–101
Griffiths TV (1968) Some remarks on derivations in general rewriting systems. Inf Control 12(1):27–54
Hotz G (1966) Eindeutigkeit und mehrdeutigkeit formaler sprachen. Elektron Inf Verarb Kybern 2(4):235–246
Huet G, Lévy JJ (1979) Call by need computations in non-ambiguous linear term rewriting systems. Rapport Laboria
Hyland JME, Ong CHL (2000) On full abstraction for pcf: I, II, and III. Inf Comput 163(2):285–408
Landin PJ (1965) A correspondence between ALGOL 60 and Church's lambda notation. Commun ACM 8(89–101):158–165
Levy JJ (1978) Reductions correctes et optimales dans le λ-calcul. University Paris, Paris
Longo G (2005) The cognitive foundations of mathematics: human gestures in proofs and mathematical incompleteness of formalisms. In: Okada M (ed) Images and reasoning. Keio University Press, Tokyo, pp 105–134
Manna Z, Vuillemin J (1972) Fix point approach to the theory of computation. Commun ACM 15(7):528–536
Meyer RK (1987) Curry's philosophy of formal systems. Australas J Philos 65(2):156–171
Milner R (1977) Fully abstract models of typed λ-calculi. Theor Comput Sci 4:1–22
Morris JH (1968) Lambda-calculus models of programming languages. Massachusetts Institute of Technology, Cambridge
Nielsen M, Plotkin G, Winskel G (1981) Petri nets, event structures and domains, part I. Theor Comput Sci 13:85–108
Pitts A (1997) Operationally-based theories of program equivalence. In: Pitts AM, Dybjer P (eds) Semantics and logics of computation. Publications of the Newton Institute, Cambridge University Press, Cambridge, pp 241–298
Plotkin GD (1977) LCF considered as a programming language. Theor Comput Sci 5:223–257

Plotkin GD (2004) The origins of structural operational semantics. J Log Algebr Program 60–61:3–15
Plotkin GD (2004) A structural approach to operational semantics. J Log Algeb Program 60–61:17–139
Primiero G (2019) On the foundations of computing. Oxford University Press, Oxford
Reynolds JC (1975) On the interpretation of scott domains. Symposia mathematica. Syracuse University, Syracuse, New York, pp 123–135
Scott DS (1969a) A theory of computable functions of higher type. In: Informally distributed, notes for a november 1969 seminar, Oxford university press
Scott DS (1969) A type-theoretical alternative to ISWIM, CUCH, OWHY. Theor Comput Sci 121(1–2):411–420
Scott DS (1970a) Constructive validity. In: Laudet M, Lacombe D, Nolin L, Schützenberger M (eds) Symposium on automatic demonstration, Lecture Notes in Mathematics, vol 125, Springer, Berlin, pp 237–275
Scott DS (1970b) Outline of a mathematical theory of computation. In: Proceedings of the fourth annual princeton conference on information sciences and systems, Department of Electrical Engineering, Princeton University, pp 169–176
Scott DS (1972) Continuous lattices. In: Lawvere FW (ed) Toposes, algebraic geometry and logic. Springer, Berlin, pp 97–136
Scott DS (1982) Domains for denotational semantics. In: Nielsen M, Schmidt E (eds) Automata, languages and programming, ninth international colloquium. Springer, Berlin, pp 577–613
Scott DS, Strachey C (1971) Toward a mathematical semantics for computer languages. In: Fox J (ed) Proceedings of the symposium on computers and automata, Polytechnic Institute of Brooklyn Press, New York, pp 19–46
Seldin JP (2011) Curry's formalism as structuralism. Log Univers 5(1):91–100
Stark E (1989) Connections between a concrete and an abstract model of concurrent systems. Mathematical foundations of programming semantics. Springer, Berlin, pp 53–79
Strachey C (2000) Fundamental concepts in programming languages. High Order Symb Comput 13(1/2):11–49
Tarjan RE (1983) Data structures and network algorithms. Society for Industrial and Applied Mathematics, Philadelphia, USA
Tymoczko T (1979) The four-color problem and its philosophical significance. J Philos 76(2):57–83
Wittgenstein L (1936) The Brown Book. Blackwell, Oxford
Wittgenstein L (1956) Bemerkungen über die grundlagen der mathematik. Blackwell, Oxford