Journal of Mathematical Sciences: Advances and Applications
Volume 1, Number 1, 2008, Pages 91-108

AN APPROACH TO THE DESCRIPTION OF FORMAL LANGUAGES' SEMANTICS

CRESCENZIO GALLO

Dipartimento di Scienze Economiche
Matematiche e Statistiche
Centro di Ricerca Interdipartimentale Bioagromed
Università di Foggia
Largo Papa Giovanni Paolo II
1-71100 Foggia, Italy
e-mail: c.gallo@unifg.it

Abstract

The creation of new programming languages, capable of fully deploying the new technological innovations and operating environments, requires more and more accurate and affordable analysis. In this study, a technique for the generation of formal models for the specification of the semantics of programming languages is presented. Tools are used newer than the semantics of Kleene - such as the Scott's theory of the categories and mathematical theory of the computation, which are here outlined and motivated.

1. Introduction

In the last years the need of a theoretical structure, suitable for the resolution of the issues arising from the formal analysis and the specifications of the semantic aspects of high-level programming languages, led researchers and analysts to focus on the development of a

2000 Mathematics Subject Classification: Primary 18CXX; Secondary 68QXX.

Keywords and phrases: formal languages, semantics.

Received April 17, 2008

© 2008 Scientific Advances Publishers

new *denotational semantics theory of programming languages* which, among the several approaches that the study of programming language semantics has known over the time, is surely the most meaningful one.

Semantics, on the other hand, provides the meaning of programs written in a particular language. This, in mathematical terms, means considering semantics as a function with a syntactically correct program as input and the description of the function computed by the program itself as output. The main players in this theory are **Scott** and **Strachey** who (together with their colleagues from the University of Oxford) demonstrated that, despite the complexity and variety of modern programming languages, it is possible - through a small number of fundamental semantic constructs - to provide adequate conceptual basis to determine short formal models about the meaning of programming languages.

There are many advantages behind the analysis of the semantic structure of programming languages. The main feature of a formal definition of such languages undoubtably is the possibility, which comes from this definition, to have a precise and complete standard reference useful for users and those implementing a particular language, to avoid omissions, contradictions and ambiguities typical of informal semantic descriptions as the historical ones of Algol '60 [10]. Furthermore, in order to determine rigorous and precise definitions demonstrations of semantic properties of languages, greatly help using a structure of the semantic concepts both general and language independent, even for standardizing terminologies, clarify similarities and differences among languages.

The usefulness of a language descriptor goes well beyond the possibility of discovering unwanted limitations, incompatibilities or ambiguity. Indeed, a general notation for describing semantic could allow the development of a real compiler generator, the same way the **BNF** notation which led to the development of analytical generators. Many of these objectives and potential of the semantic analysis of programming languages have not been achieved, even though - on the other hand - the continuing studies and subsequent progresses allow thinking that this theory will play a decisive role in the development of computer science.

2. Basic Concepts

One of the achievable objectives through the theory of denotational semantics is to demonstrate the possibility of defining the semantics of programming languages using essentially the same approach of mathematical logic. In the latter, in order to specify a semantic interpretation of a formal language, maps from the syntactic constructs of the object language in their abstract meanings are defined in an appropriate mathematical model. For example, a class of numerals could be interpreted by mapping every possible numeral in the number it denotes. Similarly, if the object language is that of the predicate calculus, every well-defined formula could be mapped into a truth value (*true or false*) on a domain interpretation and meanings specified for constants, functions and predicates. You can define the semantics of programming languages using essentially the same approach: in fact, even if the abstract meanings - proper for a programming language - are more complex and less familiar than the truth values and the numbers treated in mathematical logic, they are certainly not less mathematical.

To demonstrate the real possibility of such an approach - and to establish some notations and methodological conventions - symbols of a particular domain are used, which act as a metavariables on sets of variables, expressions, commands and programs [8]. These sets are ultimately *syntactic categories*, and the domain metavariables represent arbitrary elements of the corresponding syntactic category. A category can therefore be seen as an abstraction of "sets and functions", where the sets are called "C-objects" and represent abstract entities with no internal structure.

In this analysis it is also necessary to introduce the concept of *morphism and isomorphism*.

Definition 1 (Morphism). Given two structures $S(X, T)$ and $Q(Y, *)$, with two different operations T and * respectively on the sets X and Y and an application $f : X \to Y$, this application is said *morphism* if, denoting with a and b two elements *of* X for which $f(a)$ and $f(b)$ are the two corresponding elements of Y, we have $f(aTb) = f(a) * f(b)$, i.e., to the

result of the composition of two elements of X with the law T, in Y we have the result of the composition of the two corresponding elements with the law $*$.

Definition 2 (Isomorphism). A morphism $f : X \to Y$ in a category C is an isomorphism if there exists $g : Y \to X$ such as $gf = id_X$ and $fg = id_Y$ or, in terms of commutative diagram:

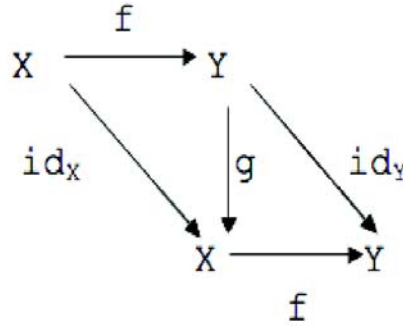

Figure 1. An isomorphism.

Let us observe that such g, if exitst, is unique; in fact, if another morphism h should exits with:

$$hf = id_X \text{ and } fh = id_Y$$

then

$$g = gidy = g(fh) = (gf)h = id_X h = h.$$

When this unique g exists, it is called "inverse of f" and is listed with f^{-1}. Two objects X and Y in a category C are isomorphic if there is an isomorphism $f : X \to Y$. This is denoted by $X \cong Y^{-1}$. In order to define a category C, let us suppose a collection of C-objects $X, Y, Z, \ldots$ and consider - for each ordered pair (X, Y) of these items - the set $C(X, Y)$ of the functions from X to Y said C-morphisms of X in Y (hereafter the term "*map*" will be used as a synonym for morphism). Let us also suppose a composition function is available which associates to each ordered pair of morphisms (f, g) of the form $f : X \to Y$ and

$g : Y \to Z$ a third morphism of the type $gf : X \to Z$, with the same domain of f and codomain of g. You can then define a category C if the above defined elements and objects are subject to the following three axioms:

1. All possible sets of type $C(X, Y)$ are disjoint;

2. The composition function is associative;

3. For each object X there is an identity morphism $id_X : X \to X$ with the following property: for every morphism $g : Y \to X$ is $id_X \, o \, g = g$ and for every morphism $f : X \to Y$ is $f \, o \, id_X = f$.

An example category is the set of subsets of *N* (natural numbers) with all the partial functions of *N* in *N*.

With regard to the concept of isomorphism, note that it is practically an equivalence relationship on the objects of a category *C*, for which it does not necessarily always exist, i.e., it is to be verified that if - taken two items in a category and a map between these objects - there is really an opportunity to determine the outputs starting from entries to the map and vice versa. This concept leads, as it were, to the *duality law*, which helps in solving problems that arise in the design or verification of a programming language. The implication of this law in the theory of categories stems from the fact that, in the latter, every theorem or deduction meanwhile is in effect because there is a dual theorem which is a demonstration of the original one, obtained by reversing the meaning of “arrows”. This property of the categories is one of the most interesting advantages over other computational techniques, just because it sometimes allows to determine constructs and semantic theorems starting from terminal elements in order to produce semantic constructs backward. We note, therefore, that stating the “isomorphism” of two objects implies the existence, between them, of all the above said conjectures.

As regards the meaning interpretive functions are defined of three types, whose codomains should be constructed according to the meaning of the corresponding syntactic class; the interpretive functions will be used to determine a path from the syntactic constructs to their

mathematical meanings. These three types of functions are called *partial functions*, *total functions* and *multi functions* and are defined as follows.

Definition 3 (Partial function). Given two sets X and Y, and considered a subset A of X, a function f connecting every element of A to only one element of Y is called *partial function* (or *partially defined*).

We will say that X is the domain of such a function, while A is the definition domain, indicated with $DD(f)$.

Definition 4 (Total function). If $DD(f) = X$, i.e., if the definition domain of f coincides with the whole domain X, then f will be said a *total function*.

The set of all partial functions of X into Y will be denoted with $Pfn(X, Y)$. In other words, a function f is partial if, taken $(x_1, \ldots, x_n)$ inputs, it is not defined for some x; otherwise, if it is defined for each input $(x_1, \ldots, x_n)$, it is said a *total function*. The set of all total functions of X into Y will be denoted with $Tot(X, Y)$. For each set X, the identity function of X is the total function $id_X : X \to X$ such that $id_X(x) = x$ for every $x \in X$.

Definition 5 (Multifunction). *A* multifunction from X to Y is, instead, a total function defined from X to all subsets of Y.

The set of all multifunctions from X to Y will be denoted with $Mfn(X, Y)$. For every function $f \in pfn(X, Y)$ the function f^o is defined in $Mfn(X, Y)$. as:

$$f^o(x) = \begin{cases} \{f(x)\}, & \text{if } x \in DD(f), \\ \varnothing, & \text{otherwise}. \end{cases}$$

Every program has therefore only one meaning given by the interpretive function of the programs which delineates, for every possible input, the output that will be produced at the end of the implementation.

The evaluation of an expression is more complex, because it requires to take account of the state of a variable when the expression is evaluated. An expression will have a unique value for each possible state;

each state represents the current value of a variable. You can then define a function S, having domain Var and codomain N, outlining each of these states. Consequently, the interpretive function for the commands will be the status of the transaction it specifies.

Definition 6 (Initial object). This is the case then to specify that an object A in a category C is initial if, for each object X in C, there is exactly one morphism from A to X. Such unique morphism is denoted with:

$$!: A \to X.$$

In this respect there is the following:

Theorem 1. *If A and B are both initial objects in a category C, then* $!: A \to B$ *is an isomorphism.*

Then, if C has an initial object, it is unique to a single isomorphism.

Definition 7 (Terminal object.) An object A in a category C is terminal if for all $X \in C$ there exists only one C-morphism from X to A, denoted with [7]:

$$!: X \to A.$$

Therefore, when a program of the type "**read variable; execute command; write result"** is executed, then the implementation must fulfill the following steps:

1. Establish an initial state in which all the variables are initialized;
2. An element of the definition domain is read and stored;
3. The body of the program runs and at the end it will be in a final state;
4. The expression is evaluated with regard to the final state, and the latter is the output.

Moreover, in case of iterative construction, the formal definition gives accurate answers about when and how often the control function should be evaluated, and about the conduct that this function must take if the value is zero.

Note that the definition of categories does not require any special constraints on the deployment of the language, as would have an implementation-oriented model, because nothing is specified about how functions should be computed and represented. Truly, for the definition of programming languages' semantics, you need only a simple procedure to assess the correct mathematical function, and this is what more can be done appropriately in a "standard" specification. The role of the only operational models of languages is simply to formalize methods for the implementation of the language, so that the accuracy of the latter can be verified through a reference to the definition of the standard [9]. What more can be done to help the verification of programs written in a language taken into consideration, is to catalog the useful deduction rules for the language constructs, as indicated by Hoare and Wirth [3].

In conclusion, the semantic analysis of a programming language is based on its denotational definition, but including on the one hand formal models of deployment, and on the other implementations of "surface"- so to speak - properties of the language constructs, and more abstractly of deeper theorems about the language in its entirety.

3. Expressions and Environments

Once we made the notion that the semantic interpretation of an expression defines its value, and that dealing with expressions so-called "pure" only the value is semantically important, it's easy to understand how it is possible that a sub-expression can be replaced by any other expression having the same value, without this change having any effect on the whole value. This linguistic property is defined *referential transparency* and the languages or subsets of languages having this feature are said *applications* [11].

We must however pay attention to the full concept of expression. It is much deeper than most programming languages can let one think: in them, the only forms of expression recognized as such are the atomic constituents (constant identifiers, etc.) and combinations *operator-operand* in the various syntactic working rules. The sub-applications, in fact, also include other forms of expressions typical of mathematical dissertations, such as:

- let $x = 5$ in $x + 3$;
- let $f(x) = x + 3$ in $f(5)$;
- $f(5)$ where $f(n) = \begin{cases} 1, & \text{if } n = 0, \\ n \cdot f(n-1), & \text{otherwise}. \end{cases}$

These involve the concept of tying an identifier to a denotation; all this corresponds to the various forms of local declaration in programming languages, which consists in the declaration of local variables, definitions of functions, formal parameter lists, iteration controlling the variables and so on. The use of bounded construction leads in general to evaluating an expression over an *environment*, which provides a value for each free variable of the expression.

The concept of *abstraction* is important for the study of expressions. In Church's notation an abstract expression takes the form $\lambda\, I.E$, where I is an identifier (the bounded variable) and E is an expression (the body usually containing I). Informally, the value of $\lambda\, I.E$ (in a given environment) is the function mapping an argument value, to which it is applied, to the value of E relative to the extended environment linking I to the argument. For example, in any environment $\lambda x.0$ denotes the constant function of value 0, $\lambda x.x$ denotes the identity function, $\lambda x.x^2$ denotes the square function, and $\lambda x.x + y$ denotes the function whose result is the sum of its argument and the value of y in that environment.

In order for the value of an abstract expression to be a function, it may look like the operator part of a combination operator-operand. For example, one can rewrite:

$$f(5) \text{ where } f(x) = x^2$$

as

$$(\lambda x.\, x^2)(5).$$

Although abstract expressions are not used in programming practice, they play a key role in the semantic analysis of programming languages. So far, as we have seen, it was not specified any particular base of

interpretation domain, but simply was assumed the existence of a values' space expressible for E and an interpretation function λ. Now, in order to determine the value of an expression which may contain identifiers generally free, it is necessary to know the values to which these identifiers are linked in that context. The set of indentifier associations and their denotations in any context is called *environment*.

4. The Mathematical Foundations

There are some mathematical problems raised by the fact that the semantic models make typically use of *higher* order functions, which make the definition of the interpretive function of expressions not as simple as we have seen previously. These functions are those whose arguments are whether functions or other infinite objects. A concrete example is given by the interpretive function of the expressions, which has an environment function as an argument. Examples of computational phenomena easily constructed using higher order functions are procedures, whose parameters or results are procedural (the flow of inputs and outputs of a not ending program as an operating system, or a common program inserted in a loop) and reentrant data structures.

One of the reasons for which the higher order functions in the semantic model create problems from the mathematical point of view is the need for *recursive definition* [4]. The traditional approach to specify the mathematical meaning of a recursively defined function (due to Kleene and other scholars) is to demonstrate that there is a partial function on a not numerable domain which is the only limit of a sequence of partial functions, each of which is at least as well-defined as the preceding items in the sequence. Of all the partial functions that could satisfy the equation defined, this limit is the least defined and is also the "natural" solution from the computational point of view. In other words, the taken solution is the smallest of the "upper bounders", meant as all the functions that could satisfy the defined equation. These solutions are also called *fixed points* of the recursively defined function. The less defined fixed point is the only function that has this property and therefore it is said *minimum fixed point* [20]. **Kleene** in one of his most important results showed that each recursive program P has a single

minimun fixed point represented with f_P. The problem with this approach comes at a time when it must be generalized to handle recursive definitions of functionals whose arguments and results can be even partial functions or functionals, or other infinite and recursively defined objects.

Another problem in applying **Kleene's** theories arises from the possibility of self-application of higher order functions, such as procedures applied to themselves of the type $f(f)$. These are those self-activating (i.e., dynamically reentrant). The problem, then, arises from the indiscriminate use of self-applicable functions, which leads to paradoxical contradictions of simple theories. A classical example is given by a predicate which is true when its argument is a predicate that is false when it is applied to itself: that is to say, if $p(q)$ is true when $q(q)$ is false, then we would have that $p(p)$ is true if $p(p)$ is false, which is absurd.

In essence defining semantics becomes more complicated if you must interpret recursive functions, because defining the "meaning" of recursion is not so obvious, as in the case of instructions as "**if-then-else**". It is useful, therefore, to better define the meaning of recursive function.

Definition 8 (Recursive function). Let X and Y be sets. A recursive specification for a function $f \in Pfn(X, Y)$ is a function Γ such that the value of $f(x)$ (recursive specification) is obtained starting from x and from a finite number of values of f by function:

$$\Gamma(f(x) = \Gamma(f(0), \dots, f(n))).$$

The problem arises: "what is the semantics of Γ?" The goal is to establish a function f such $f = \Gamma(f)$; at this point f will b the semantics relative to the recursive specification Γ.

A recursive specification defines therefore a function in terms of itself. Sometimes, however, there are examples of recursive specifications which show that the desired denotational semantics is not always entirely clear. It can indeed happen that the recursive specification is seen as "an equation". You can then have situations where its solution is total and

unique, but it can also happen that its equational solution is not unique. If the recursive specification is seen instead as an algorithm who calls itself, then there can be more than one algorithmic solution, depending on the calling strategies; besides, not always an algorithmic solution has an equational one or, again, there are algorithms that do not terminate.

A mathematical theory of computation providing satisfactory solutions to these problems has been developed by **Scott** using concepts of mathematical logic and topology. The basis of **Scott's** theory lies in the fact that this characterizes classes of data models, called *domains*, and classes of functions (including the high order ones) general enough to allow natural models of computational phenomena (including *recursion* and *self-application*) but also sufficiently restricted by a number of axioms in order to avert all theoretical paradoxes and allow finite approximations. These axioms are justified by showing that mathematically consistent spaces and semantic models can be constructed satisfying the same conditions. In other words, the main feature of **Scott's** domains is that a sequence of better and better approximations, in a domain, must converge to a limit that best lends itself within the same domain. In order, however, to "protect" these limitations, the operations defined on the data model must be *continuous* (this concept is much more general than that of the analysis just after the application scope was defined).

The *primitive domains* must be formed by adding to finite or not numerable sets as $\{true, false\}$ or $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ two special symbols $\perp$ and $\top$, respectively, called "bottom" and "top". The first represents the completely undetermined information also called *initial*, while the later represents the consistent or entirely determined information. The primitive sets considered are:

$N = \{\ldots, -2, -1, 0, 1, 2, \ldots\}^{o}$ (whole numbers),

$T = \{true, false\}^{o}$ (truth values),

$H = \{'a', 'b', \ldots\}^{o}$ (characters),

where $\{\ldots\}^{o}$ denotes the addition of the special symbols $\perp$ and $\top$. In such domains the notion of *approximation* is really simple: $\perp$ "approximates" all elements and all elements "approximate" $\top$, while all other pairs are incomparable. Therefore, there are not trivial limits or recursive definitions of elements in primitive domains, while the added domain structure is needed just to meet the general demands of axioms and provides a basis for the construction of more complex domains.

A large number of nonprimitive domains can be built through appropriate transactions. In fact, if D, D_1 and D_2 are domains, then the following are also domains:

i. $D_1 \times D_2$,

ii. $D_1 + D_2$,

iii. $D_1 \to D_2$,

iv. $D_n = D \times D \times \cdots \times D$,

v. $D^* = D_0 + D_1 + D_2 + \ldots$.

Apart from the special treatment of $\perp$ and $\top$, the elements of $D_1 \times D_2$ are ordered pairs whose first components are elements of D_1 and whose later components are in D_2. One element of $D_1 + D_2$ corresponds to an element of D_1 or D_2. The domain $D_1 \to D_2$ consists of continuous functions from D_1 to D_2. D_n and D^* are, respectively, domains of tuples and of all finite fists of elements of D. Each of these built domains also contains special elements $\perp$ and $\top$ and, in some cases, even partial items with approximation relations of the constituent domains, derived from them. For example, in the case of a domain made of functions $D_1 \to D_2$, f approximates g when $f(x)$ approximates $g(x)$ for all $x \in D_1$. Several constructions can be combined into a domain definition, Sintactically it is assumed that the operator of binary domains "×" has the highest priority and "→" the lowest (and it associates at right as above).

Returning to the concept of continuity, it can be said that constants and identity functions on each domain are continuous, and that any function defined through abstractions and combinations is continuous only if its sub expressions are still continuous on the domain. It is important to note, moreover, that on primitive domains the requirement of continuity comes down to monotonity concept.

Definition 9 (Monotonity). We can thus say that a function f is monotone when, if x "*approximates*" y, then $f(x)$ "*appromimates*" $f(y)$.

Each partial function f on a set can then be extended to total continuous functions on the corresponding domain defining $f(x) = \perp$ (since $\perp$ approximates each element), $f(\top) = \top$ (since $\top$ is approximated by each item) and $f(x) = \perp$ if the partial function is undefined in x.

These extensions are designated "*doubly strict*" (doubly rigid). It is possible, however, to have less strict extensions of functions when a function is constant in comparison to one of its arguments: then you do not need to have $\perp$ as a result, even if that argument is undefined.

We can now deal with the above mentioned problem related to the specification of the mathematical meaning of a recursive definition. This problem consists in finding a fixed-point function Y_D that produces an appropriate solution to equations of the form $f = F(f)$, given the higher order transformation $F : D \to D$. It is clear that if D is a domain, then there is an approximation relation on it and a starting element $\perp$; so, using monotonity we have that $F(\perp)$ approximates $F(F(\perp))$, and inductively:

$$\perp, F(\perp), F(F(\perp)), \ldots, F^i(\perp), \ldots$$

is a sequence of better and better approximations which, due to continuity, converge to a limit f such that $F^i(\perp)$ approximates $f \; \forall i \geq 0$ and $F(f) = f$.

It was considered a special induction technique called *fixed point induction*. This technique is a powerful tool that can be used to prove assertions about the minimum fixed point of functions [6].

You can also show that for every domain there is a continuous function of fixed point $Y_D : (D \to D) \to D$ such that, for each continuous $F : D \to D$:

1. $Y_D(F) = \lim_{i\to\infty}(F^i(\bot))$ of successive approximations, is a solution for the equation $f = F(f)$;

2. each other solution of the equation is approximated by $Y_D(F)$.

This result is a generalization of **Kleene's** theorem of classical recursion [4] in which the approximation relation allows both arguments and results of recursively defined functions to be higher order partially defined objects, instead of strictly defined or undefined. The arguments can be generalized to give meaning to arbitrarily complex systems of mutually recursive definitions. The problem of self-application is resolved, too, by the **Scott's** theory showing that both domains and domain's items can be recursively defined.

This brings our brief excursus on **Scott's** theory of computation to an end. It has been shown how this theory solves the problems raised by the higher order interpretation and how you can then proceed on the analysis of increasingly complex languages with the assurance that the resulting semantic models are mathematically correct, as long as one only deals with functions and domains defined using the bove-stated method. The semantic idea introduced (environments, categories, sequences of approximations) provides a conceptual structure for the formal semantic specification of almost all configurations of high level programming languages, and stands as conceptual basis to address issues such as indeterminism, time compilers [23] and more complex control structures as backtracking, coroutines and parallelism.

5. Conclusions

Nowadays, because of the continuing need to develop and design formal tools able to exploit the great potential that the technology offers, analysts and industry experts increasingly become aware that there may not be an adequate progress in the field if there is not a solid conceptual substrate consistent and not ambiguous. For this purpose the availability

of conceptual tools as Scott's denotational semantics of programming languages is important which, through all its interpretations and abstractions, is an approach that can solve the most complex formal aspects easily and naturally.

This work is intended to highlight the impact that **Scott's** theory had and may have on the description of the semantics of programming languages. The denotational theory was examined, starting with the ideas and basic concepts that led **Scoot'** and **Strachey** to develop the conceptual construct starting from abstract algebra, mathematical logic and set theory.

It was used, to the purpose, an abstract concept of operation which, starting from any object (initial or not) provides a certain result. It was subsequently introduced a more general concept of set, apart the proper one of the set theory, in which such an interpretation is not always orthodox (in fact, the set theory almost always implies restrictions on the types, prohibiting to consider operations whose domain consists of all objects). It outlines as a useful approach to avoid the inconvenience that may be experienced in dealing with more abstract concepts, rather derived from the habit to think in set terms coming from the objective difficulty to formulate a mathematical system based on a less restrictive concept of operation.

This is why a formulation of a doctrine based on the theory of categories was given, despite its fundamentals are not yet fully well developed. It was intended to give greater emphasis to the **Scott's** formulation - in connection with his research on the semantics of programming languages - on the possibility of giving a more sophisticated interpretation of semantic models in which the terms are intended as a succession of functions and none of them remains without interpretation (the **Scott's** general theory of models and his motivations in terms of "computer science" are published in [13]; these models have led to very interesting results, even though most of them has not yet been published).

We also led ourselves to a different resolution of the problem of recursion, based on a mathematical rather than computational

interpretation of recursive definitions, describing the essential idea of considering the recursive definition as a *correct* mathematical equation by replacing the concept of "is" (classic of a computational approach) with the "=" one for the equivalence relations. This approach allows you to think of recursively defined functions as fixed point functions of an higher type compared to the fixed point functions used by **Kleene**, which first provided an elegant treatment of recursion. Of great practical importance is that this mathematical approach has also led to the discovery and use of a powerful induction rule to prove the observations of certain recursively defined functions.

This discussion, ultimately, although formulated in terms of a general and unspecified programming language, wanted to express the basic idea of the theory of denotational semantics of formal languages and indicate its potential about solving the fundamental problem of any recursive program: how to describe its precise meaning. It can hardly be called a program, let alone the language that defines it may be so, until this issue is not defined.

The objective of this approach is meant to prove, finally, the existence of an appropriate balance between strict formulations, the breadth of applications and conceptual simplicity. The essential purpose is to show that, insisting on appropriate level of abstraction and using the correct mathematical rules, it is possible to frame a method which can be described as "*the mathematical meaning of a language*".

References

[1] L. Allison, A practical Introduction to Denotational Semantics, Cambridge University Press, Cambridge, 1986.

[2] F. S. de Boer, R. M. van Eijk, W. Van Der Hoek and J.-J. C. Meyer, Failure semantics for the exchange of information in multi-agent systems, C. Palamidessi, ed., Eleventh International Conference on Concurrency Theory (CONCUR2000), University Park, PA, 2225 August, number 1877 in LNCS, pp. 217-228, Springer-Verlag, 2000.

[3] C. A. R. Hoare and N. Wirth, An axiomatic definition of the programming language Pascal, Acta Inf. 2 (1973), 335-355.

[4] S. Kleene, Introduction to Metamathematics, Van Nostrand, New York, 1952.

[5] P. J. Landin, The mechanical evaluation of expression, Computer J. 6 (1964), 308-320.

[6] Z. Manna, Mathematical Theory of Computation, Mc Graw- Hill, 1974.

[7] E. G. Manes and M. A. Arbib, Algebraic Approaches to Program Semantics, Springer-Verlag, 1986.

[8] A. Meyer and S. Cosmodakis, Semantical Paradigms, Proc. Third Annual Symposium on Logic in computer Science, pp. 236-255, Computer Society Press, 1988.

[9] R. E. Milen, The formal semantics of computer languages and their implementations, Ph.D. Th., Cambridge U. and Tech. Microfiche TCF-2 Oxford U. Computing Lab., Programming Research Group, 1974.

[10] P. Naur, Revised report on the algorithmic language Algol 60, Comm. ACM 6(1) (1963), 1-17.

[11] W. V. Quine, Word and Object, Technology Press, Cambridge, Mass. and Wiley, New York, 1960.

[12] J. C. Reynolds, Notes on a lattice-theoretic approach to the theory of computation, Dep. Systems and Information Science, Syracuse U., Syracuse, New York, 1972.

[13] D. Scott, Outline of a mathematical theory of computation, Proc. 4th Princeton Conf. on Information Sciences and System; anche Tech. Mon. PRG-2, Oxford U. Computing Lab., Programming Research Group, 1970, pp. 169-176.

[14] D. Scott, The lattice of flow diagrams, in Engeler 1971; also Tech. Mon. PRG-3, Oxford U. Computing Lab., Programming Research Group, 1971, pp. 311-366.

[15] D. Scott, Continuous lattices, Proc. 1971 Dalhousie Conf. Springer Verlag Lecture Note Series, n. 274, Springer—Verlag, Berlin, Heidelberg, New York; also Tech. Mon. PRG-7, Oxford U. Computing Lab., Programming Research Group, 1971.

[16] D. Scott, Mathematical concepts in programming language semantics, AFIPS Conf. Proc., Vol. 40, 1972 SJCC, AFIPS Press, Montvale, N. J., 1972, pp. 225-234.

[17] D. Scott, Lattice theory, data types, and semantics, in Rustun, 1972, pp. 65-106.

[18] D. Scott, Lattice theoretic models for various type-free calculi, Proc. 4th International Cong, for Logic, Methodology, and the Philosophy of Science, Bucharest, 1972.

[19] D. Scott, Data types as lattices, Unpublished lecture notes, Amsterdam, 1972.

[20] D. Scott, The Kleene Symposium, North-Hollan Publishing Company, 1980.

[21] D. Scott, Lambda Calculus: some Model, some Philosophy, J. Barwise, H. J. Heisler and K. Kunem, eds., 1980.

[22] D. Scott and C. Strachey, Towards a mathematical semantics for computer languages, Proc. Symp. On Computers and Automata, Polytechnic Institute of Brooklyn; also Tech. Mon. PRG—6, Oxford U. Computing Lab., 1971, pp. 19-46.

[23] R. D. Tennent, Mathematical semantics and design of programming languages, Ph.D. Dep. of Computer Sci., U. of Toronto, 1973.