Pubblicazioni del

CENTRO STUDI CALCOLATRICI ELETTRONICHE
del C. N. R.

presso l'Università degli studi di Pisa
Some Preliminary Remarks on Theoretical Pragmatics
By A. Caracciolo*

The term pragmatics has been used, as is well known, by Morris and Carap to denote that branch of semiotics which deals with the relation between a language and its users. As such I don't like the term very much. But one might also define it to be that branch which deals with the relation between languages and actions or processes. This seems more appropriate to me. In any case, whatever the definition, there is a basic connection between programming languages and pragmatics, as pointed out by Gorn. I devoted a paragraph to this subject in my talk at the IFIP Congress 1965 entitled "Linguistic Problems in Programming Theory."

The basic reason for this connection between programming languages and pragmatics is that programming languages are not simply formal referential languages, like those devised by logicians and other theoreticians for defining and referencing this or that type of abstract mathematical entity. Rather they are languages to be used by programmers for communicating to a job executor what job it has to perform. And it is essentially this concept of a job, which deals with actions and processes, which justifies the word "pragmatics" in referring to this field of research.

This concept of a job, or task, is indeed basic to programming language theory, as has been recognized many times in the present conference, and as I have tried also to show in my IFIP 65 talk. There I proposed, in order to begin a clarification of the concept, to classify tasks into concrete vs. abstract, ending vs. non-ending, and sequential vs. parallel and concurrent types. It is difficult to define what a "job" is, mainly because it is not an isolated concept but acquires a meaning only in a larger context. The concept of a job, in fact, requires that there exist some job executor capable of performing the job, and that there exist a description specifying what the job is. Actually the concept of a job and the concept of programming language are very intricately interrelated, as can be seen if we try to define "programming language."

A programming language is a language to be used by a programmer for communicating to an appropriate job executor what job it has to perform. The basic concept therefore appears to be all of these concepts fused together, and this is what we have been calling an "operating system" at this conference.

However, much of the discussion has been essentially empirical, revolving around a number of undefined concepts, and, what is worse, these concepts were not well understood not only by the audience, but, at least in some cases, by the speakers as well.

An important task therefore appears to be to try to give some basic formal definitions of the concepts involved. This, and simply this, is what I understand by "theoretical pragmatics." It is a difficult task; however, I consider it a solvable problem which deserves a great deal of attention. In the following I shall try to indicate a possible line of attack which I believe might prove quite useful.

The preceding definition of programming language has many implications: first, that the programming language must be known both by the programmer and the job executor; second, that the programming language must be capable of expressing both what the programmer wants done and what the job executor is capable of doing. Moreover, it implies that there exist both programmers, who want to do something and are capable of expressing what they want in some appropriate way, and a job executor capable of accepting a job description and performing the actions required. It is more than clear that such things exist, but the problem still remains of trying to define them in order to be able to make best use of them. This last point has been the main theme of this conference; it is what one could call "applied pragmatics," which, however, only makes sense if there is a "theoretical pragmatics" to be applied.

This conference has considered not only programming languages which are in a sense unidirectional (from programmer to job executor), but also bidirectional conversational languages to be used in continuous man-machine information exchanges. A clarification of such languages would require the clarification of the concept of a conversation. This seems a rather difficult task, especially if a conversation is to be mixed up with performing at the same time one or more jobs. I shall therefore limit myself here to unidirectional programming languages.

* University of Pisa, Italy.
Let me try to illustrate by means of an example a possible approach. I shall choose for this purpose the concept of a “programmed machine tool operating system.” The basic concepts we shall use are rather similar to those used in Automata Theory, from which they differ essentially in the effort of taking time into explicit consideration. As a matter of fact this approach is very similar to that discussed by Elgot both in his work with Robinson on random access stored program machines, and in his Vienna conference talk on machine species and their programming languages.

Let us first define what is understood by a “programmed machine tool.” Although we are oversimplifying and are not completely rigorous, we define a programmed machine tool as a system $M(X, Q, \nu, T(m), \gamma, \rho)$ where $X$ is a set of possible input states, $Q$ is a set of possible machine tool states, 

$$T(m) = \{ t_0, t_1, \ldots, t_n \},$$

$\nu : X \times Q \rightarrow Q$ is a basic mapping characterizing the machine tool from a functional point of view, $\gamma$ and $\rho$ are two mappings

$$\gamma : X^T \times Q \times T \rightarrow Q,$$

and

$$\rho : X^T \times Q \rightarrow Q.$$  

(Where $X^T$ denotes the set of functions from $T$ to $X$), which are defined as follows: Let $I \in X^T$ denote any member of $X^T$, and let $\phi : X^T \times T \rightarrow X$ denote that mapping which for each $I$ and $t$ gives the value associated with $I$ at time $t$. Then $\gamma$ is defined by means of the recursive equations

$$\gamma(I, q_0, t_0) = q_0,$$

$$\gamma(I, q_t, t_{i+1}) = \nu(\phi(I, t_i), \gamma(I, q_t, t_i))$$

and $\rho$ simply by

$$\rho(I, q_t) = \gamma(I, q_t, t_{i+1}).$$

From the definition of $\gamma$ one could also easily get a mapping

$$\beta : X^T \times Q \rightarrow X^T$$

which defines for each initial state $q_0 \in Q$ and each input $I \in X^T$ the “behavior” of the machine tool as a mapping $\chi : T \rightarrow Q$.

We could continue by introducing a high-level machine tool programming language in the context of a high-level programmable machine tool operating system, for instance, as a system

$$\langle \mathcal{C}, \mathcal{L}, \tau, \Gamma(H, L, \omega), \mathcal{C} \mathcal{L} \rangle,$$

where $\mathcal{C}$ is again a formal language, $\mathcal{L}$ is a machine tool oriented programming language for $\mathcal{C}$, $\tau$ is a recursive mapping of $\mathcal{C}$ into $\mathcal{L}$, and $\Gamma(H, L, \omega)$ is a computing machine capable of transforming a concrete string of characters belonging to some set $H$ into a concrete string $L$ under the control of a machine state $\omega$ (a program corresponding to the mapping $\tau$) such that

$$\omega_{[\tau(h)]} = q_{[\tau(h)]},$$

where $h$ is any linguistic expression belonging to $\mathcal{C}$, and $q_{[\tau]}$ and $q_{[\tau]}$ are two mappings

$$\omega_1 : \mathcal{C} \rightarrow H,$$

$$\omega_2 : \mathcal{L} \rightarrow L$$

which connect formal languages with concrete strings related to the computer.

We can now proceed to define a “programmable machine tool operating system” as a system $\langle \mathcal{L}, \mathcal{C}, 3, \rangle$, where $\mathcal{L}$ is some formal language and $3 : \mathcal{C} \rightarrow X^3$ a mapping which relates a formal language to the complete set of signal time series acceptable by $3\mathcal{L}$, and therefore, through $\beta$ to the behaviors of $3\mathcal{L}$, defines a basic pragmatic relation between the formal language $\mathcal{L}$ and the machine behaviors. It is only in such a context (i.e., embedded in a programmed machine tool operating system) that we call $\mathcal{L}$ a “machine-tool-oriented programming language.”

All this might appear to be somewhat clumsy. However, the basic reason seems to be due not so much to the proposed approach as to the fact that we are dealing with a complex set of interrelated notions. In my opinion, the clumsiness of the approach simply reflects, and therefore clarifies the situation. The real problem, therefore, apart from the obvious one of making progress in building effective operating systems, appears to be the need for a careful analysis by the suggested scheme or some analogous one. We should try to simplify such conceptual schemes, of course, but not by permitting confusion by over-simplification, questionable generalization, or illegal identifications.

**DISCUSSION**

**Gorn:** Are you assuming that the motion of the machine tool itself is quantized in time in the same way that this formalism indicates?

**Caracchio:** In this scheme, yes.

**Gorn:** Here is what I would consider a pragmatic question. How would this formalism distinguish a machine tool which moves in a circle with a quantum of time from one which moves in a straight line in that same time? You have exactly the same formalism for describing the system.

**Caracchio:** Yes. It’s equivalent. It all depends on what you mean by $Q$, which gives the states of the machine; and how do you get from one station to another station except with the tool?

**Gorn:** So the actions must be specified when we describe the way the time is quantized.

**Caracchio:** Of course. I wanted to show how to design a programming language for machine tools. But this means one has to formalize somewhat. The machine tool is something which, accepting certain inputs, performs certain acts to go to another state from a preceding state.

**Buchman:** Your machine description tells how you accomplish the transformation of $X$, $Y$ and $Z$ coordinates of the machine tool. That is what varies your $3\mathcal{L}$ function.

**Caracchio:** Yes. The $L$-language can tell what inputs the machine tool accepts. There is a function, $\nu$, a structural mapping characterizing a special machine. The mapping corresponding to the machine’s behavior is $\nu$; when $\nu$ is presented with an input which is a function of time, it causes changes of states in the machine. Here I considered time to be discrete. The model could be generalized to permit time to be continuous.

**Now:** Is this not equivalent to the description I suggested the other day: Output $= f$(input). It describes all possible programs in the world. Isn’t it almost the same thing?

**Gorn:** But slightly more elaborate.

**Caracchio:** No. It is more elaborate in that input is only one aspect; the sequencing of machine states is also in the description. Time is taken into account. And $\Gamma$ gives a mapping into an output function.

**Kirsch:** In the definition of $3\mathcal{L}$ as an abstract machine, I can’t see any special properties that justify your calling it an automatic machine tool control.

**Caracchio:** That is right. To get a machine tool, really, one must specify $Q$, $X$ and $\nu$. I haven’t done it. But this shows exactly what must be done.

**Kirsch:** But that is true of any machine; this gives you a wide class.

**Caracchio:** That is a good point; it is a scheme which can be applied to many situations.

**Kirsch:** My objection is that $3\mathcal{L}$, though perfectly well defined, is, among other things, the definition of some of my friends.