

THE SEMANTICS OF PARALLEL PROCESSING

H. Bekic

Rapporteurs: Mr. G.M. Arnold
Mr. I. King
Mr. P.M. Melliar-Smith

Summary

Dr. Hans Bekic developed a mathematical technique for formal definition of the semantics of combinations of elementary actions, and extended this to encompass parallelism by determining the fixed points of infinite sequences of such actions.

Dr. Bekic began by pointing out that, although IBM Vienna Laboratories was doing work on parallelism, he intended to cover aspects of the semantics of programming languages, but with some emphasis on parallel programs.

From the start of the work on programming language semantics, the main approach was the use of abstract machines in what was described as the "constructive way" of language definition. In this one considers the states of the language interpreting machine, where ξ_0 (the initial state) is determined by an abstract version of the program and its input data, and where the iterative application of the interpreting function yields successive states $\xi_1, \xi_2 \dots$ until either termination is reached or the computation proceeds forever. A particular feature of the PL/I definition is that the state transition function was allowed to be non-determinate, or, to express it in better understood terms, it is not a function from a set of states to a set of states but, rather, from a given state it produces a set of possible answers, and is thus a function from a set of states to a set of subsets of states.

Dr. Bekic continued by saying that since the above work his views of descriptive semantics had been much influenced by the work of Landin and of Scott, and that in his recent work he had moved in the direction of mathematical semantics. However the problem of non-deterministic programs remained and formed the basis of the material

to be presented; namely, can one have a mathematical view of semantics, and can one deal with nondeterminate functions in a way that captures the underlying pragmatic notions.

Dr. Bekic first indicated briefly why the constructive technique of language definition failed to cover intuitive notions of associating meanings with expressions in a language. Drawing the simple analogy of a purely descriptive language of arithmetic expressions, one associates with each expression a certain value: thus, in a language which uses Roman numerals, the value of each numeral, V, X, I, is a number, 5, 10, 1. Extend the language to include composite expressions, and write (in list notation)

$$+, (e_1, e_2)$$

then by associating values with the expressions e_1 and e_2 , the resulting value is a function of those values. In particular, if the expressions e_1 , e_2 involve identifiers, or are identifiers, then one must know from outside, or from context, what the values of these identifiers are going to be.

Thus is introduced the notion of environment, a function from identifiers to whatever one chooses as values. The interpreting function now takes not only an expression but also an environment, and yields a value. Although this is a trivial example, it indicates a difference in approach compared with the earlier work; there is no reason why the idea of environments giving meaning to identifiers cannot be carried over to algorithmic languages. Although expressions will denote either numbers, or more complicated things such as transformations of the machine state having more complicated values, the interpreting function will still take an expression, or statement, and an environment of the appropriate form, and will produce a value for the expression or a more complicated result for the statement.

It is necessary to distinguish between identifiers, which are associated with values, and denotations, which are the objects associated with programs or subprograms. As the programming language probably contains assignment statements, one has a notation of store, which maps storage locations to values, and the denotation of an assignment statement will just be a transformation from stores to

stores. Furthermore, one may arrange things in such a way that the denotation of composite statements depends only on the denotations of the simpler components. In this way one can represent the input-output behaviour of the program, and in so doing abstract from many things that might be considered irrelevant, so reducing the complexity of the machine state. However this also abstracts from details that for some purposes one might be interested in; in considering programs as expressions of algorithms, it may be essential to be able to analyse two different programs evaluating the same function using different algorithms. In other words there is a notion of denotation that covers more than just the function computed by an algorithm, namely the steps by which the function is computed.

When analysing Algol 68 using the denotational approach there is a problem that the "collateral composition" of functions does not yield a further function; that is, if states transformations are considered to be composed of several individually indivisible steps, the collateral composition of two such step sequences is a non-deterministic operation (if one merges the two sequences there ~~may~~ be many outcomes computed by the complete set of mergings). This complicates the chosen denotations in two ways. Firstly, rather than being simply state transforming functions they must be at least sequences of such functions. The second complication is that, due to the element of non-determinism, the denotation of a given expression might be a set of such sequences. Dr. Bekic added that in the latest definition of a large subset of PL/1 in the "new style", the complications due to non-determinism and parallel interaction have been left untouched.

Denotation

The main purpose of the presentation by Dr. Bekic was to introduce the formal notations of denotation and to indicate how mathematical semantics can be employed, particularly for solving recursive equations for functions. In what follows, denotations are treated as if they are functions. Although such functions are state transformations, programs do not in general use a variable for the state. For instance, in serial composition

$f ; g$

the expansion of which is the function

$\lambda \xi . g(f(\xi))$,

the variable ξ is never used in the program.

A useful combinator is that which besides changing the state also yields a value - in programming usually termed an "expression with side-effects". This is written

let $v : e ;$
... $v ;$...

Here e is a state transformation which also yields a value, while v is just a state transformation

$e : X \rightarrow V \times X$
 $v : X \rightarrow X$.

For this combination one may write

$\lambda \xi . \text{let } \langle v, \xi' \rangle = e (\xi)$
... $v (\xi')$

which explains the combinators of simple let notation. To include combinators such as parallel composition

$f \parallel g$

it is necessary to reinterpret all objects as more complex objects and reinterpret the combinators in terms of these more complex objects.

Such a language of combinators has proved to be quite convenient for representing given source language programs, so that from a PL/I or Algol 68 program one may derive a particular denotational expression using these combinators. This may be done by an extended version of the interpret function by which the semantic correspondence is defined. Such a derivation is a static process which, rather than executing the program, produces the corresponding meta-language program in terms of " ; ", " : " and " let ".

A more complicated notion of denotation is called "action", after a related notion defined in the Algol 68 report. Using the idea of "action" and the notion of "hand-translating" source programs into meta language programs it is possible to formulate some notions of compiler correctness.

In the questions that followed, Dr. Bekic confirmed that his talks were covering the "new style" of language definition in which the new definition of PL/I had been carried out. He added that although there had been changes in the meta language, the most important change was away from the idea of an interactive machine and towards the association of meanings with expressions. Questioned about the example he had given of an expression with side effects, he said that the notation

let v : e

described a declaration which allocated storage and returned the location for subsequent use; e is basically meant to return a value, but in doing so it changes the state. Following further questions he added that the "let" binds e and ξ to the body of the function associated with v.

Actions

In his second lecture Dr. Bekic introduced the more complicated notion of actions, went on to define certain compositions of actions, and finally dealt with the problem of solving recursive equations for actions.

Rather than considering simple state transitions, one must consider compositions of such transformations from others which may in some sense be considered to be indivisible or elemental. The notion of action is based on sequences of elementary transitions, but is more complicated in that an element in the sequence determines what is done next, but the next element may be dependent on the current state as well as on the continuation. Thus is obtained the following definition of an action:

Let X be the set of computational states, and

let ξ be a state value from the set.

$A = \underline{0} + X \rightarrow ((X \rightarrow X) \times A)$

The action A is either the empty sequence $\underline{0}$ or depends on the given state ξ . Thus there is a function from X, the set of states, to a pair. The first member of the pair is a state transformation of the set, and the second member is the remaining actions to be done. Thus, if α is an action, then apply α to the state ξ :

$$\alpha(\xi) = \langle f, \alpha' \rangle, f: X \rightarrow X$$

The action determines what happens next and what is left to be done. This definition of an action is similar to the head-tail definition of a list. It is also necessary to admit the possibility of infinite actions in the sense that the above lists may become infinite. The disjunctive union sign is used here, although the sets are already disjoint, because the set is used as a definition in the same manner as Scott, in that there is an undefined element. This is still not enough, however, in non-deterministic programs, because of the possibility of choosing what to do next independently of the current state, since we may choose freely any one of several possibilities.

Deterministic Actions

Therefore, rename the set A to be the set of deterministic actions dA . An action then is just a set of deterministic actions, and the set of actions is a subset of the sets of deterministic actions.

A collection of actions can be built up from elementary items. So far, there is the null action. Let f be a state transformation which is an elementary action, then:

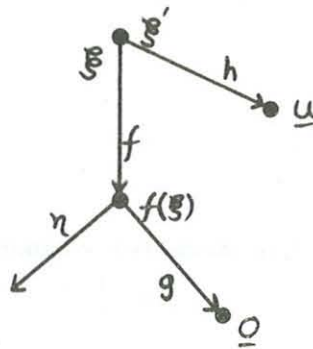
$$[f](\xi) = \langle f, \emptyset \rangle$$

Applying f to the state ξ gives a pair, namely f , as the step which is performed, and the remaining action which is null. Thus for each state transition there is a corresponding elementary action consisting of just that state transition.

Serial Composition

To define a Serial Composition of two actions, define operations on the superset of deterministic actions. Strictly, dA is not a subset of A because dA is essentially an element of A . However, elements may be identified between the sets and thus one may regard dA as a subset. Greek letters will be used for elements of dA and Latin letters for the more general actions.

The serial composition $\alpha;\beta$ of α and β devolves onto the restricted set dA . It is convenient here to picture actions as forming a rather complicated kind of tree. Complicated because there is a dependency on ξ at each node.



Choosing ξ as the current state, an action f can occur followed by a state to which action g can be applied. Eventually this process must terminate with the null action. Equally, from another state ξ' there might be the undefined action \underline{U} . Note that \underline{U} is included implicitly as a starting element.

This gives a representation of the set dA . (A is just a set of such animals.) The inductive definition of serial composition is

$$\alpha;\beta = \left(\begin{array}{l} \alpha = \underline{U} \rightarrow \underline{U}, \\ \alpha = \underline{O} \rightarrow \underline{\beta}, \\ (T \rightarrow \lambda \xi. \text{let } \langle f, \alpha' \rangle = \alpha(\xi) \langle f, \alpha' ; \beta \rangle \end{array} \right)$$

If the first element α is undefined, then so is the result, since there can be no continuation. If the first element is null then the composition consists of the second, and otherwise there is a function from states to pairs.

The gain here, is in the recursive definition. Since the domains involved are continuous, and the elementary objects like conditional expressions are continuous, such definitions can be used to derive a continuous function from an infinite sequence. The function to be defined is a serial composition of actions, not just deterministic actions, and this is given by:

$$a;b = \{ \alpha;\beta \mid \alpha \in a, \beta \in b \}$$

the set of all compositions of deterministic actions.

One way to explain this is to choose an element of a and an element of b , and compose them, but that is not the way it is normally considered. One thinks of doing a , and, having done it, one of the many possibilities has been realised.

Parallel Composition

Consider first the parallel composition of deterministic actions:

$$\alpha \parallel \beta$$

but the result of this must be a non-determinate action, and conditions will again be recursive. Postpone the problem of recursive definition of functions using sets, because of the ordering problems. It is not obvious how ordering relations are to be defined, so sets will not be introduced in this definition. Rather, let the definition depend on a third parameter, written as an index, giving a three case function, taking two deterministic actions and this hidden parameter. The hidden parameter can be an infinite tape of choice values: $t \in T$, $T = \{0,1\} \times \mathbb{T}$.

Define next $(\alpha \parallel \beta)_t$.

First the simple cases:-

$$(\alpha \parallel \beta)_t = \begin{pmatrix} \alpha = \underline{0} \rightarrow \beta \\ \beta = \underline{0} \rightarrow \alpha \\ \alpha = \beta = \underline{U} \rightarrow \underline{U} \end{pmatrix}$$

If the first action is null then the composition comprises only the second, and vice versa. If both are undefined, so is the result.

This is compatible with the previous definition. Moreover if α is undefined and β is not, commencing with α stops with an undefined outcome. However, commencing with β may make it possible to continue. Only when α is performed is the action undefinable. This is where the choice parameter is used. If the next token on the tape U_t is zero, continue with the next part of α and only then go on with parallel composition. Introducing a new operation here, $\alpha \overset{\uparrow}{\parallel} \beta$ "left parallel" indicating that, without choice, the first step of α is executed first, the definition is completed with

$$T \rightarrow (U_t = 0 \rightarrow (\alpha \overset{\uparrow}{\parallel} \beta)_{t_1 t}, U_t = 1 \rightarrow (\beta \overset{\uparrow}{\parallel} \alpha)_{t_1 t})$$

where U_t signifies the use of the tail of the choice value tape.

$(\alpha \parallel \beta)_t$ is defined as follows:

$$(\alpha \parallel \beta)_t = \left(\begin{array}{l} \alpha = \underline{0} \rightarrow \beta \\ \alpha = \underline{U} \rightarrow \underline{U} \\ (T \rightarrow \lambda \xi. \text{let } \langle f, \alpha' \rangle = \alpha(\xi) < f, (\alpha' \parallel \beta)_{t_1 t} \rangle) \end{array} \right)$$

Thus when α is not null or undefined, the composition is a more complicated action, namely a function transforming ξ the current state into a pair, the result of the first step of α , and the parallel composition of the remainder of α with β under the tape, having used the first token on the tape.

This definition is still very restricted compared to that in existing programming languages; namely a state is indivisible, a single entity, and it is known of the functions only that they transform that entity. There is no notion of parts of a state. If there was such a notion, then parallel composition could be defined in a more direct way, without trying to mix or describe all possible sequences produce the same effect. Even with such more complicated states, there will also be parallel compositions which act on a part of the state, and it will still be necessary to decide on their meanings.

The operation $a \parallel b$ can now be specified as the set of all $(\alpha \parallel \beta)$ over T .

$$a \parallel b = \{(\alpha \parallel \beta)_t \mid \alpha \in a, \beta \in b, t \in T\}.$$

Consider further two very simple, compositions:

$$a \text{ or } b$$

Since the actions are defined as sets. This is just set-union

$$a \text{ or } b = a \cup b.$$

Also, a conditional,

if p then a else b , where p is a predicate on states. To show the testing of p as a separate step, define it as follows:

$$\text{if } p \text{ then } a \text{ else } b = \{\lambda \xi. \langle I, (p(\xi) \rightarrow \alpha, T \rightarrow \beta) \rangle \mid \alpha \in a, \beta \in b\}$$

$$\text{and } p : X \rightarrow \{T, F\}$$

It is an action of functional type, and its first component is the identity function, the next step being either a or b depending on the state.

Recursive Definitions

Define an action by

$a = \text{if } p \text{ then } 0 \text{ else } f; a$

or by $a = f \vee (g; a; k;)$.

The usual technique for solving such equations is to start with a as undefined, and in the general case $a = F(a)$. Thus, starting from \underline{U} and iterating F , we form a limit to the sequence obtained. Since the objects which are obtained are sets of determinate actions an ordering between such sets is needed. It is easy to define an ordering on determinate actions. If the branch of the action tree for one action α ends in undefined whereas for another action β at that point there is a continuation possible, then β is more defined than α , written $\alpha \equiv \beta$.

This relation could be extended to sets, to introduce

$$a \equiv b = \forall \alpha \in a \exists \beta \in b \alpha \equiv \beta$$

and conversely. Thus, a is less defined than b if for all α in a and β in b α is less defined than β , and conversely, for all β there is an α less defined than β . Then considering the case $a = \{\alpha, \gamma\}$ and $b = \{\alpha, \beta, \gamma\}$, suppose $\alpha \equiv \beta \equiv \gamma$. Then if $a \equiv b$ there is a continuation, β as a continuation of α . But equally, if $b \equiv a$ there is a continuation, since β and α may continue to γ . Thus the relation is not an ordering.

Thus the whole idea of using sets as the universe over which to solve the recursive equations is wrong.

An alternative approach uses elements and the notion of the hidden parameter. Consider the following examples. Take

$$a = (fvg) \parallel h; a.$$

This example has no conditionals, and thus the determinate actions can be considered as sequences of functions. Starting with \underline{U} and applying $F(a)$ to it as above gives a whole set of possibilities, and thus a whole family of functions dependent on hidden parameters. Given any deterministic action α , and taking the right hand side above, first prefix it by an h and insert an f or a g at some point. (That point might be infinitely distant, since it cannot be assumed that in parallel execution either one of the two actions is necessarily

performed within any specific period of time. One of the two actions may be repeated for ever.) This gives the following family of functions:

$$F_{\phi}^n(\alpha) \quad \phi \in \{f, g\}, \quad 0 \leq n \leq \omega$$

ϕ is either f or g , and n ranges from zero to ω . For a given n and ϕ , F is defined as follows:-

$$F_{\phi}^n(\alpha) = h; \alpha \quad \text{with } \phi \text{ inserted after } n \text{ elements.}$$

Consider the restricted case of actions as sequences of functions, then, given an n and ϕ , this definition is not complete. Some provision must be made for n larger than the length of the sequence. The undefined case needs to be corrected. Thus the insertion is not made after the n th element, but after $\min(n, l(h; a))$ elements if possible where $l(h; a)$ is the length of the sequence. Anything that would have been inserted after undefined will not be inserted because sequences ending in undefined are neutral over composition. This ad hoc definition of a family of functions, covers all possibilities of non-deterministic operations.

The finding of a fixed point of the original equation can now be approached, in the deterministic case, by applying functions iteratively. Since there is a choice of which functions to apply, consider sequences of the form

$$F_{\phi_1}^{n_1}(\underline{U})$$

and replace \underline{U} by the application of further such functions forming

$$F_{\phi_2}^{n_2}(\underline{U}) \quad \dots \quad F_{\phi_k}^{n_k}(\underline{U}).$$

Then form the limit of that sequence:

$$\text{Lt}_{k \rightarrow \infty} F_{\phi_1}^{n_1} (F_{\phi_2}^{n_2} (F(\dots(F_{\phi_k}^{n_k}(\underline{U}))))))$$

Now that limit is for a given sequence of n, ϕ values.

$$\text{i.e. } \langle n_1 \phi_1 \rangle, \langle n_2 \phi_2 \rangle \dots \in (\bar{N}_0 \times \{f, g\}^\omega)$$

where \bar{N} is the set of numbers between zero and ω . Thus, the choice tapes may be considered as presenting n, ϕ pairs instead of truth values, in order to determine the choice of function.

These generate a set, a , of all infinite sequences of elements h, f, g .

$$a = \{ \{h, f, g\}^\omega \mid \#f + \#g \leq \#h + 1 \}$$

Each time an f is applied, it is certainly prefixed with an h , and there may or may not be one more f or g inserted, thus forming all sequences having for every subsequence the number of f 's plus the number of g 's, less than or equal to one more than the number of h 's.

Actions with Choice Nodes

Rather than prove of the above example either that the set of limits really gives the required set a , or that the construction really solves the original equation, it is of more interest to know for a certain class function F , how generally, one can devise such a family of non-deterministic functions and use this same method.

To do this, introduce another space of objects, called actions with choice nodes. Above, a tree represented actions of a certain kind. Now introduce a more complex tree in which branches are labelled with zeros and ones to denote possible alternatives at each choice. Thus all possible paths are contained in the one tree.

The equation for this set will be:-

$$A = \underline{0} (x \rightarrow x) \times \hat{A} + \{0,1\} \rightarrow \hat{A}$$

and again compositions similar to those above can be defined, for example $\hat{a};\hat{b}$ in which the second tree can be appended to any end of the first. In the case of $\hat{a}||\hat{b}$ there are the same trivial cases for null or undefined, but in the other case, a new tree forms dependent upon the choice of zero or one. In the case of one, take the next step of \hat{b} .

Define a function "range of"

$$r : \hat{A} \rightarrow A$$

\hat{A} gives all the possibilities required, and in addition to A gives the labelling of particular subtrees with zeros and ones. What is needed is to leave out the labelling information, so the range of \hat{A} , r , is the set of paths in the tree disregarding labels. This rather imprecise definition, could be given a precise inductive form. The crucial property of

r is that it is not a continuous function, since there are no continuous functions from elements to sets, but it has the useful property of compatibility with the various compositions. For example

$$r(\hat{a};\hat{b}) = r(\hat{a});r(\hat{b})$$

This allows, given some equation $a = F(a)$, the forming of the corresponding equation on the set \hat{a} , $\hat{a} = \hat{F}(\hat{a})$, which can be solved because the \hat{F} 's are continuous and so are their compositions. The fixed point may be formed by the usual construction $\hat{a} = Y \hat{F}$ and a step made into the other space using $a = r\hat{a}$ where a solves the original equation because

$$F(r\hat{a}) = r(\hat{F}\hat{a}) = r\hat{a}$$

and because \hat{F} is compatible with all the functions and therefore with their composition F , and of course \hat{a} is the fixed point of \hat{F} .

An Example

For his third lecture, Dr. Bekic considered the example of a very simple equation for an action a:

a is the effect of an action f in parallel with a

$$a = f \parallel a$$

Certainly this is not deterministic. Using the technique above he introduced a family of functions F_n of a deterministic action α which insert f after the n'th position, with a qualification on the insertion that f can not be inserted beyond the undefined symbol regardless of n.

The limit of the elements of a is the set of limits $F_n, F_{n_2}, \dots, F_{n_k}$ of u, where these run over all infinite sequences of the modified integers containing the element ω as above.

F_0 is the operation of inserting f at the beginning of the sequence. An infinite number of such operations causes an infinite number of fs. In all other cases the sequence will be terminated by the undefined symbol, as in the sequences:

$$\begin{array}{c} \underline{u} \\ f \underline{u} \\ f^2 \underline{u} \\ \vdots \\ \vdots \end{array}$$

and also the sequence f^ω

Then Dr. Bekic introduced an alternative approach to the definition of the semantics of such parallel processing making use of the set \hat{A} of actions with choice nodes and with trees in which these choices are recorded. The choice tree can be applied to a particular tape to obtain family of actions.

$$\underline{ap}: \hat{A} \times T \rightarrow dA$$

$\underline{ap}(\hat{a}, t)$ selects a subtree of \hat{a} , according to the tape t , by deleting edges from the tree.

The function $\underline{r}(\hat{a}) = \{\underline{ap}(\hat{a}, t) \mid t \in T\}$ yields all possible values of applying all different tapes.

Dr. Bekic further considered the equation

$$a = F(a)$$

to define the family of functions F_t .

$$F_t: \hat{A} \rightarrow dA$$

$$F_t: [dA \ \underline{c}] \hat{A} \rightarrow dA$$

F_t of \hat{a} is the result of the operation on the choice tree producing a further choice tree which can be applied to a given tape. Thus the type of the function is a choice tree to a family of functions. The determinate actions are themselves a subset of the choice trees, so that although F_t is a function over \hat{A} it is certainly a function over a subset.

In the case of this particular example

$$F(a) = \{F_t(\alpha) \mid t \in T, \alpha \in a\}$$

Thus $f||a$ here is just the operation already discussed of $(f||\alpha)_t$.

The definition of parallel composition already required a corresponding additional parameter, and the expansion of this determines valid conditions for $f||a$ to have a meaning. The functions F_n, F_t are therefore the same but indexed differently, one by tapes and the other by integers, and the function F_t can be used to form the set of limits.

Dr. Bekic demonstrated that this simple relationship between F and F_t does not hold even for the restricted class of functions formed by compositions of primitive combinators. Consider an equation formed from the combinators but with a present more than once, say k times.

$$a = F(a) = G(a, \dots, a)$$

Assume that G satisfies a similar equation

$$G(a_1, \dots, a_k) = G_t(\alpha_1, \dots, \alpha_k) \quad \left| \begin{array}{l} t \in T \\ \alpha_i \in a_i \end{array} \right.$$

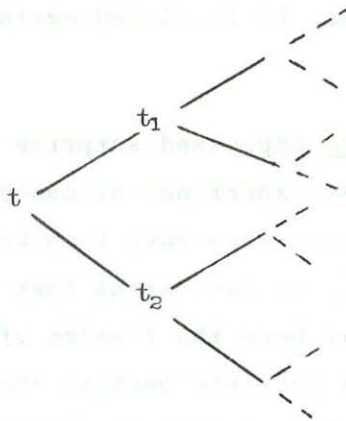
Then the relationship is not true because different α_i could appear for different occurrences of α .

Representable Functions

However the relation is valid for functions which are just compositions of the functions defined with the combinators. For such functions a family of functions G can be derived

$$G(\hat{a}_1, \dots, \hat{a}_k) = \underline{ap}(\hat{G}(\hat{a}_1, \dots, \hat{a}_k), t)$$

obtained by first applying \hat{G} to the choice trees and then applying the result to t . To solve this equation, we must take not an infinite sequence of t 's, but a tree of t 's. Start with t , and if that t has two arguments, take two additional t 's, and so on. These infinite trees must be approximated by finite trees which have undefined to terminate each branch.



Then

$$a = \left\{ \begin{array}{l} \lim \\ t \end{array} \right. G_t(G_{t_1}(\dots(\underline{u})\dots), G_{t_2}(\dots(\underline{u})\dots))$$

where $\left. \begin{array}{l} t \\ \left. \begin{array}{l} t_1 \\ t_2 \end{array} \right\} \right\} \in T$

There is no proof yet that the fixed point of this construction is the same as the fixed point $\sim (\gamma\hat{F})$ derived above. To derive such a proof it would be necessary to use the fact that instead of using just one tape, the tape can be split up and different parts used for different portions of the evaluation, and that conversely the different portions of the tape represent a pattern that exists on the whole tape.

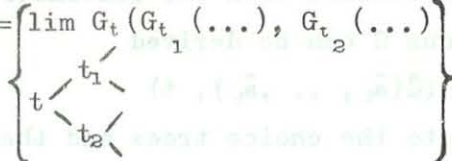
A more interesting question relates to what is it that makes the function F have a fixed point.

For a deterministic function, the function just has to be continuous, but for a non deterministic function there can be no notion of continuity.

If $F:A \rightarrow A$ has a representation like

$$f(a) = \{G_t(\alpha_1, \dots, \alpha_k) \mid t \in T, \alpha_1 \in a_1\}$$

If there is one fixed point of F

$$a = \text{fix } F = \left\{ \lim G_t(G_{t_1}(\dots), G_{t_2}(\dots)) \right\}$$


then we can call such a function a representable function. It can be shown that parallel composition, serial composition, and the OR operation are representable and that their compositions are again representable, so that this class of functions is closed against arbitrary substitution. It is closed against fixed points.

Discussion

Professor Dijkstra expressed surprise at this approach, in comparison with 15 years experience of non-determinacy. The techniques developed by experience have been transferred to the writing of sequential programs, to the extent that non-deterministic sequential programs may have the freedom of a generalised petri net. But this had only been possible because the specific choices have no effect on the result, and this is absolutely essential because of the very large number of possible sequencing choices. If the parallel computation is to be meaningfully useable then it is necessary that the greater part of the non-determinacy is absorbed at the earliest possible stage, before its effects diffuse through the computation and are difficult to destroy.

Dr. Bekic replied by considering a large program containing a small part, or several small parts, which can be highly non-deterministic. Yet the program can be arranged in such a way that this non-determinacy vanishes so that, however the inner parts correlate to each other, it can be proved that the whole program is determinate. Nevertheless, to establish this, it is necessary to

model the behaviour of the inner programs.

The limit construction is very significant. In the examples, families of functions were defined where the choice phase was not critical because the choices are all finite. From a certain n on there is no difference, and all F_n from n on are the same for a given α . Thus the choice basis reduces to the family F_n , to how many functions F_n there are. Of course this is very different from the question of non determinacy propagating from the inside outwards, from small parts to large programs. There is no way to deal with that.

Professor Dijkstra considered the simple loop.

while B do S

for known B and S. The semantics of this can be defined by an inductive limit. The difference to accomodate indeterminacy in S is that, instead of considering the effect of k repetitions, the consideration is of the effect of at most k repetitions.

Professor Scott drew attention to the different choices in the ways in which parallel processes merge, and suggested that it is necessary to consider the way in which the processes have been divided into steps and the ways in which these steps can be merged. He invited Professor Dijkstra to express the meaning of his loop without such consideration.

Professor Dijkstra explained that the repeatable statement must itself be regarded as non-deterministic, so that its final state is not uniquely defined by the initial state.

Professor Scott expressed interest in how that indeterminacy propagates as the processing is performed. In a good program the non-determinacy can be absorbed, but to provide a general definition of the semantics of arbitrary programs, it is necessary to consider ill-designed programs.

Dr. Bekic asked Professor Dijkstra to determine for his loop the permitted combinations of B and S so that the loop remains determinate.

Professor Dijkstra considered the construct

DO : while B do S od

S being a nondeterministic activity. If we can form, for any post condition R, the precondition for the initial state such that an activation of S is certain to lead to a terminating operation ending with the system in state R, we can introduce a series of preconditions

$$H_0(R) = R \text{ and } \neg B$$

and for $i > 0$

$$H_i(R) = B \text{ and } \text{up}(S, H_{i-1}(R)).$$

For a deterministic machine

$$\text{up}(DO, R) = (E_k, k \geq 0, H_k(R)).$$

This as a rule is a negligible complication to the recurrence relation, and the existence of this term is the only place where the non-determinacy shows up.

Professor Scott enquired the meaning of H_0 when the precondition B is either true or false. Professor Dijkstra replied that $H_0(R)$ is the weakest precondition such that this operation will lead to a terminating operation with final state R with at most zero repetitions, but nothing can have happened and so R must hold to start with.

Professor Scott asked how the weakest precondition for S_0, s , in parallel is expressed.

Professor Dijkstra gave the example

if $x \geq y \rightarrow m := x$

|| $x \geq x \rightarrow m := y$

fi

Professor Scott asked for a general expression for $s || s'$. Where s and s' run in parallel, it was necessary to break them into atomic steps and to count and sequence all the steps.

Professor Dijkstra felt that that particular parallelism had been very imperfectly modelled by being converted into a sequential operation. The important topic is the way in which non-determinacy,

as generated for instance by parallelism, propagates through the theory.

Professor Bekic summarised, saying that in a programming language where such things can not happen, there can be no general parallel operator. There are conditons under which operations may execute in parallel, or very restricted operators. Alternatively a language may provide genuine parallel composition. Of course, in a sensible program, sensible use would be made of it, but there may be other uses of a more general character. On a higher level one would not wish the non-determinacy to propagate, but the language should not be syntactically restricted. Rather, the definition of the meaning of the language must take into account the general case, because the most complex case can be useful. The objective of considering a language with a general parallel operator and the defining of its meaning is still a valuable one.

