The University of Warwick

THEORY OF COMPUTATION

REPORT NO. 26

AWAY FROM THE OPERATIONAL VIEW OF COMPUTER SCIENCE

BY

WILLIAM W. WADGE

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
ENGLAND.

November 1978

Away from the operational view of Computer Science

William W. Wadge
Department of Computer Science
University of Warwick
Coventry England

Abstract

We argue that classical programming languages are based on a fundamentally mistaken emphasis on the operational aspect of computation. These languages are seen as the means by which the programmer brings about particular kinds of operational activity (such as procedure calling or message passing).

We suggest an alternate philosophy which places the emphasis on static, extensional (mathematical) concepts such as that of function or sequence. We outline a simple functional language (Luswim) based on the principles. The Luswim programmer specifies the desired output, and operational concepts such as data flow, message passing or coroutine linkage can be used to 'evaluate' the specification. Operational activity occupies its proper place, namely as means to an end rather than an end in itself.

Preface

This paper is based on a talk the author gave to an SRC (UK) sponsored workshop on languages for parallelism and distributed computing. The workshop took place at The University of Warwick, 25-26 September 1978. The text has been drastically condensed and edited, so that (for example) there are no incomplete and garbled sentences, or glaring examples of self contradiction. But I have preserved the informal flavour of the presentation - a good excuse to avoid the work involved in a proper paper. Since the content is primarily philosophical, rather than technical, I think this is not such a bad idea. Also, at several places I have added between parentheses sentences or whole paragraphs which were not in the original talk, even in a garbled form. Furthermore, comments from the audience have been paraphrased or omitted.

There is a short bibliography of important papers which have been included because of general relevance, and not necessarily because specific reference to them was made in the text.

My talk is only indirectly concerned with Lucid. Instead, I've chosen the more humble goal of telling you everything that's wrong with everything you're doing. (much laughter)

I'm only partly joking! These errors are best discussed in the context of everything that has been wrong with computer science for the last twenty years, and that's nicely illustrated by this chart :

syntax	semantics
efficiency	correctness
operational notions	denotational notions

These are three pairs of related yet opposite (dual) concepts. Now in each case it is necessary for us to study both concepts. But in no case are the concepts equal; one is always more important than the other; and what I'm saying is that many of our troubles and errors result from a mistaken emphasis on the wrong member of a pair.

For example, I think we would all agree that for a long time in the study of PL's there was vast over emphasis on syntax. Language definitions were primarily specifications of syntax, and whole volunes were devoted to the parsing problem.

But I would argue that semantics (what you mean) is more important than syntax (how you say it). It is only recently that with the systematic study of semantics that we are getting anywhere near solving the big problems in our subject.

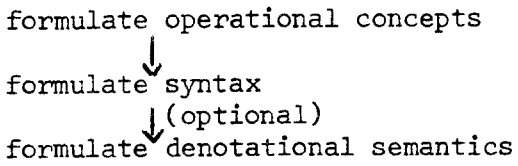
Nevertheless, the over emphasis on syntax still remains. We all know what a legal Pascal type definition is, but what is a Pascal type? Does anybody know?

Similarly, for a long time people have been obsessed with making programs work quickly. Lately, however, it has become apparent that the real problem is correctness (or reliability). Again, correctness relates to what you want, and efficiency relates to how you get it. As someone (Landin?) said, we should write correct programs and make (or implement) them efficiently, rather than write efficient programs and make them (by debugging) correct.

The obsession with efficiency remains (especially in North America, under the name "complexity") but at least our colleagues are aware of the problem. Now what I'm saying is that there is a third even deeper misemphasis which is still not widely realised as such - the emphasis on the dynamic operational side of computation (in other words how a machine computes something) rather than on the static, denotational side (what the machine is computing). (Notice that each of the three errors is an instance of emphasising means over ends.)

The primarily operational view of computation is reflected in many branches of computer science, but it causes particularly serious trouble in the area of programming language design. The current interest in parallelism has made these problems very acute, but they existed before, when we were satisfied with von Neumann machines.

I want to explain now what it means in reference to PL design methodology. There is a lot of discussion about programming methodology, but not much about PL design methodology. Nevertheless, there is a standard methodology (which I call the "classical" one) illustrated by the following diagram :



For example, we might first formulate the operational notion of a procedure call in which you 'go away' taking arguments with you, 'perform' the procedure (and this might effect the environment) then 'return' with the results. Then we cook up some syntax for procedure declarations and calls, spend a lot of time and money implementing it (and even more marketing it).

(It inevitably turns out that for mysterious reasons our operational concepts weren't quite general enough. For example, we now want call by name (and reference, and value result), we want static as well as dynamic binding, textual expansion (macros) versus calling etc., etc., etc.. We include these all and after a few iterations we are dealing with a monstrous collection of operational concepts, each with its own syntax and its own page in the manual. And yet there always seems to be some new concept left out, or some special case, or some combination.)

I missed out the third step, which I've indicated as optional because it is often omitted in practice. It consists of giving a denotational semantics to the language resulting from the first two stages. Of course this is not a job for your average computer scientist on the street; quite the contrary, only the most highly skilled semanticists steeped in the Scott-Strachey approach are up to it. Often the language designer doesn't understand the semantics when it's derived; and often he doesn't want to know, so that the whole exercise takes on a religious flavour - the language has been 'blessed' by a semanticist. (much laughter)

This is not to say that the achievements of the Scott-Strachey school are not valuable. But they are so successful (and their tools are so powerful) that the value has been negated. Their problem is that they can give a semantics to anything - no matter how complex and bizarre the operational concepts, no matter how baroque the syntax, they'll give it a denotational semantics. Recently, they gave a semantics for SNOBOL. Is this really a useful achievement? It seems to me they are just giving the language designers a blank cheque.

(At this point Robin Milner suggested that semanticists should and do provide feedback to the designers with respect to what and what is not a good idea.) Yes that happens, but it's not enough. (After all, the advice to an operationally motivated designer must take the form of saying that such and such an operational concept is bad. Yet it's a rare operational concept that can't have a case made for it, that one can't find a use for.)

What I'm saying is that the present role of denotational semantics (at best, advice giving) is inadequate. We need a qualitatively different methodology which gives extensional concepts priority. I could illustrate it with the following diagram :

formulate extensional concepts
↓
formulate syntax
↓
investigate operational semantics

For example, we might start with the extensional concept of function and then consider operational concepts (such as calling and binding conventions, etc.) which would be useful in implementing functional languages. In this context we can now talk sensibly about these different operational concepts being necessary, useful, efficient or whatever (and very few will be completely ruled out).

Tony Hoare in his talk yesterday gave a classification of languages. It was an operational classification, i.e. according to the operational activity the programmer could bring about. I'm not saying this classification is incorrect or useless, but I don't think it's the most crucial one. The most important distinction is between languages based on operational concepts, and those based on denotational (Gordon Plotkin suggested the word "extensional") concepts. On the one side we would have EPL and CSP, along with Modula and the classical sequential languages Algol, Fortran, etc.. On the other side (a much thinner crowd) we have Lucid (designed by the author with Ed Ashcroft), Turner's SASL, Kowalski's predicate logic language, and Landin's ISWIM. McCarthy tried to make LISP one of these, but didn't quite succeed (because of dynamic binding and replaca etc.).

(My message is that if we are ever going to be able to take advantage of parallelism and distributed computing, it will have to be with languages of the second type.)

To illustrate the alternate methodology, I will design you a language (called Luswim) for parallelism before your very eyes. (This language is a disguised form of Lucid, and is based on ideas developed with Ed Ashcroft.)

We begin with the extensional notion of operation (as in the addition operation) (as opposed to the operational notion of an operation "which is performed"). The simplest programs are just expressions like

$$2 * 3 + 4 - 5$$

built up from constants and operation symbols. The meaning of such a program is the value of it as an expression. Even at this trivial level, even when we are using the language of arithmetic, there still exist relevant operational concepts which are significantly different. For example, we could use a straightforward left-to-right sequential stack algorithm to evaluate expressions (i.e. leftmost innermost); or "one-off dataflow"(asynchronous parallel innermost). These are two completely different methods, but they give the same answer.

We need only the extensional notion of a sequence of values to give us, in conjunction with the syntactic notion of variable, a much more powerful notion of program. We define a program to be a set of compatible definitions. Each definition is an equation, with a variable on the left hand side and an expression on the right. The fact that they are compatible means that no variable has more than one definition. The

variables defined are the output variables, while those which are used but not defined are the input.

Here is a sample program :

$$\begin{aligned} D &= B^2-4AC \\ R1 &= (-B+\sqrt{D})/(2A) \\ R2 &= (-B-\sqrt{D})/(2A) \end{aligned}$$

whose input variables are "A", "B" add "C", and whose output variables are "D", "R1" and "R2".

The semantics of programs are defined as follows: given any values for the input variables, the corresponding values of the output variables are those determined by the equations, i.e. the solutions of the equations. Of course in general there may be circularities in the dependencies and as a result the equations may have more than one solution. In this case we must be able to choose some 'distinguished' solution - almost always the least, if our collection of data objects forms a cpo. In any event, we determine a functional correspondence between values of input variables and values of output variables, and this function is the meaning of our program.

This notion of program is very general and powerful and in introducing new 'features' we will make only minor changes, mainly expanding the class of allowable expressions.

(I should mention that I am using the word "variable" as logicians do - a variable is a syntactic object, what computer scientists also call an "identifier". Unfortunately, computer scientists often use the word "variable" to refer to a semantic concept, namely to mean a bucket or storage location which holds a value. This peculiar and nonstandard use of the word "variable" causes enormous confusion when one discusses issues such as "variable-free programming").

Just in defining this simple notion of program we have already twice made use of the fundamental extensional notion of function. Clearly, we would like to let the programmer specify his own functions. The syntactical changes are minor; we add a collection of function variables and allow equations to define functions using other variables as formal parameters, e.g.:

$$\begin{aligned} F(x,y) &= x + axy + G(x-1)/y \\ G(x) &= \sin(x) - \cos(x) \end{aligned}$$

(much as with Fortran's function definition statements). We can define the meaning of a program just as before - the least solution of the equations (important results concerning function domains make this possible).

When it comes to considering operational semantics, we definitely have our work cut out for us, especially if the definitions are circular (if there is recursion). We can adapt the left-most innermost algorithm which now corresponds to a left to right call by value implementation. The only problem is that it doesn't always work, because it may hang up (fail to terminate) when the maths (and even our intuitions) say otherwise. Even if it doesn't hang up, it's inefficient because it performs unnecessary computation.

In short, call by value is simple and has its uses but is very restricted. If we want more correct and more efficient implementations we must use some technique (like call by name) which delays evaluation - though of course there is no reason not to mix methods.

A big problem with our language as defined so far is that there is no scope ('hiding') - all variables, including intermediate results, are considered as output. Therefore we take the extensional notion of selection (of a value from a sequence of values) and incorporate it syntactically as follows. We allow 'compound expressions' of the form

prodof

$$\begin{aligned} V_0 &= E_0 \\ V_1 &= E_1 \\ &\vdots \\ V_n &= E_n \end{aligned}$$

end

where the 'body' is a program one of whose output variables is "out". The value of such an expression is the value of "out" as determined by the program constituting the body.

This idea is really a trivial variation on Landin's where-construct, and (as Landin pointed out) it gives much the same scope rules as Algol. Operationally, Algol-like methods can, in certain circumstances, be used to evaluate expressions containing such constructs: you allocate space for locals, compute their values, deallocate the space and return with the value of "out". Also, for programs with functions, an Algol-like display is necessary to keep track of references - static binding is required, and dynamic binding gives the wrong answer.

A great variety of operational approaches can be used to implement the language so far, but none of them make significant use of iteration - which is, after all, a very important operational concept. Surprisingly, this can

be remedied by using the simple extensional concept of infinite sequence (indexed by the natural numbers). Syntactically, we enlarge our language by adding two unary operation symbols "first" and "next", and a binary operation symbol "fby" (more are possible but these will do for now). Then programs have the same meaning as before but we interpret them over the domain of sequences of data objects. Also, our new operations are defined on sequences as follows :

$$\text{first } (\langle x_0, x_1, x_2, \ldots\rangle) = \langle x_0, x_0, x_0, \ldots\rangle$$
$$\text{next } (\langle x_0, x_1, x_2, \ldots\rangle) = \langle x_1, x_2, x_3, \ldots\rangle$$
$$\langle x_0, x_1, x_2, \ldots\rangle \text{ fby } \langle y_0, y_1, y_2, \ldots\rangle = \langle x_0, y_0, y_1, y_2, \ldots\rangle$$

and all other operations work 'pointwise' (these are of course the basic ideas behind Lucid).

Then the following equations:

$$I = 1 \text{ fby } I + 1$$
$$J = 1 \text{ fby } J + 2I + 1$$

define J to be the sequence of squares.

There are many ways to implement such programs; they could be translated into an iterative imperative language; or into a data flow net; or run on a call-by-need interpreter. Also, there is no harm in the programmer using one of these operational semantics as an (informal) guide, because it is correct - gives the same answer (under certain conditions).

My time is nearly up and I've so far described an applicative language with its own form of expressions, assignments, scope, procedure declarations and iteration; in other words with all the features of Algol. It seems I haven't even started to talk about distributed computing which, after all, is what this workshop is all about.

Yet there is in fact no need to extend the language just defined. In particular, there seems to be no need at this point to formulate an extensional notion of "process". In considering operational interpretations of programs in the language just as it is, a good many 'distributed' operational concepts (such as dataflow, coroutines, message passing with or without queueing, dynamic process generation and so on) suggest themselves naturally and forcefully. This rich variety of possibilities results "free of charge" from the interaction of sequences and the other features of the language.

As a first example, consider this program which defines Y to be the average of the corresponding initial values of Y:

```
X = -----
Y = prodof
      N = 1 fby N + 1
      S = X fby S + next X
      out = S/N
    end
```

The variables "N" and "S" are local to the prodof phrase but the phrase cannot be understood operationally as an Algol block, because we must imagine that their values are kept around from one 'invocation' to the next (N is a counter, and S keeps a running total of the values of X so far). Instead, we have to see it as a continuously operating "process" or "actor" with local memory, repeatedly taking in values of X and repeatedly producing (hence "prodof", not "valof") corresponding values of Y.

Next we present a program in which we 'abstract' this block into a function and then use it:

```
avg(X) = prodof
           N = 1 fby N + 1
           S = X fby S + next X
           out = S/N
         end
I      = 0 fby I + 1
S2     = avg (I)
S3     = avg (I)
out    = S2 * S3
```

Then in implementing programs like these we treat the body of a function definition like that of "avg" as a template, calls of which produce separate instances each with their own copies of the private local variables. Notice that this is not the same as an Algol procedure with local own variables - when (as above) the function is used in two different places, the internal values corresponding to different calls would get tangled up with each other, as a result of which we would get the wrong answer.

Next, consider the following program (due to Gilles Kahn) which defines S to be the stream of all numbers of the form $2^i\ 3^j\ 5^k$ (in increasing order):

```
 S = 1 fby merge3(S2,S3,S5)
S2 = 2*S
S3 = 3*S
S5 = 5*S
```

where merge3 is a 3-way merge (which expects its input to be increasing sequences and returns their ordered merge with repetitions eliminated). This program (and the merge3 function) are very easy to understand operationally from the data flow point of view but queueing is required if you want the right answer.

This doesn't mean that all programs which can be implemented by data flow require queueing. The need for queues arises because of certain functions(like merge3) which consume their arguments at a different (e.g. slower) rate than they produce their results. Once we know which functions to watch out for, we can, given a program, isolate those arcs on which queueing might take place.

Finally, consider this prime generating program which (in various forms) is becoming a classic:

```
N = 2 fby N + 1
out = sieve (N)
sieve(M) = prodof
            MO = first M
             D = M whenever ¬ MO divides M
           out = MO fby sieve(D)
           end
```

Again, it can be understood in terms of dataflow provided you allow nets which grow dynamically (or in terms of processes and message passing, provided you allow the dynamic creation of processes).

Again, we can pinpoint the reason for the need of dynamic creation - the fact that sieve is recursively defined. As before, this knowledge can be used in a "stress analysis" of programs.

I hope these examples at least indicate that the ideas I have been discussing are very relevant to distributed computing. (I would add that, from my point of view, "Languages for distributed computing" was not the best title for the workshop because it implies that we use languages to bring about such-and-such a mode of distributed computing. Rather, we should be interested in languages (like the one described) which can be usefully implemented using distributed techniques - in other words, in languages which can use distributed computation.)

(There was a lengthy discussion after the talk the basic point of which was that sometimes (e.g. in real time systems) behaviour is part of our ends, not just of means. This is true (although exaggerated), and would genuinely require extensions of the language defined here -

e.g. sequences indexed by the reals, not the natural numbers. However, the simple sequences mentioned above can be used to specify behaviour to a far greater extent than is realised (and also behaviour needs specification to a much less extent than is widely realised). For example, the behaviour of an interactive editor can be specified as a function relation between the sequence of user inputs and the sequence of computer outputs).

A short bibliography

Ashcroft, E.A. and Wadge, W.W.
Lucid, a nonprocedural language with iteration,
CACM 20, No.7, pp. 519-526.

Clauses : scope structures and defined functions
in Lucid, Proc. POPL 77.

Kahn, G.
The semantics of a simple language for parallel
programming, Proc. IFIPS 74, pp. 471-475.

Kowalski, R.
Predicate logic as a programming language
Proc. IFIPS 74, pp. 569-574.

Landin, P.J.
The next 700 programming languages,
CACM 9, 3(1966), pp. 157-164.

Vuillemin, J.
Correct and optimal implementation of recursion in a
simple programming language, 5th Annual ACM Symposium
on Theory of Computing, Austin, 1973.