INFORMATION PROCESSING 77, B. GILCHRIST, EDITOR

COROUTINES AND NETWORKS OF PARALLEL PROCESSES

GILLES KAHN
Iria-Laboria
Rocquencourt, France

DAVID B. MacQUEEN
University of Edinburgh
Edinburgh, Scotland, U.K.

The concept of *coroutine* or *process* is useful in a large class of applications, usually involving incremental generation or transformation of data. We present a language based on a clear semantics of process interaction, which facilitates well-structured programming of dynamically evolving networks of processes. These networks exhibit the same input/output behavior whether they are executed sequentially or in parallel. Sample program proofs are used to illustrate the benefits of the language's simple denotational semantics. The language serves also to clarify the relationships between coroutines, call-by-need, dynamic data structures and parallel computation.

1. INTRODUCTION

Many algorithms are naturally organized as systems of independent processes which coexist and interact with one another. In this paper, we present a structured approach to the programming of such systems. This approach is embodied in a programming language which subordinates control to structure, relieving the programmer of the burden of control management and permitting process systems to be executed either sequentially or concurrently with the same result. The language was designed to reflect the clear semantic conception of process interaction presented in [1], with the result that programs are relatively easy to verify.

1.1 Coroutines, multipass algorithms, and pipelines

Our notion of a *process* is derived from Conway's original concept of coroutines, [2] which he introduced as an improved way of executing multipass algorithms. In his words, "... *a coroutine is an autonomous program which communicates with adjacent modules as if they were input and output subroutines.*" The coroutines represent successive passes each of which *incrementally* transforms a stream of data, so that their execution can be interleaved in time according to a "demand-driven" scheduling strategy. This mode of execution was described succinctly : "*When coroutines A and B are connected so that A sends items to B, B runs for a while until it encounters a read command, which means it needs something from A. The control is then transferred to A until it wants to write, whereupon control is returned to B at the point where it left off.*"

Conway went on to note that coroutines can be executed *simultaneously* if parallel hardware is available. This is possible without time-dependent side effects because the coroutines communicate with each other only via input/output instructions, and this in turn follows from the fact that the coroutines are modelling separate passes of a multipass algorithm. When executed in parallel, such a system is called is a "pipeline." Many studies of parallel program schemata have been concerned with generalizing this simple linear organization.[3-5]
The classic illustration of coroutines is the cooperation between the lexical analyzer and the parser in a compiler. However, algorithms structured as a set of interacting coroutines occur in many applications besides compiling, such as input/output handling, [6] text manipulation, [7] algebraic manipulations, [8] sorting, [9] numerical computation, [10] and artificial intelligence. [11] The UNIX operating system [12] provides a "pipelining combinator" in its command language which is used to connect programs together in linear pipelines for quasi-parallel execution.

1.2 Alternative approaches to coroutines

A different approach to coroutines, typified by the SIMULA control primitives *call*, *detach* and *resume*, is fairly widespread. [13-14] The SIMULA primitives can be used to *implement* Conway's style of coroutines, where control transfers are hidden in the input/output commands, but they also allow many other types of interaction which often result in intricate control relationships. Use of the *resume* command in particular leads to obscure control structures, because it resembles a *go to* command with a moving target. For the sake of program reliability and verification one needs to impose discipline on the use of these primitives, and when this is done [16-18] it leads to the structuring of process interaction along the lines of Conway's original proposal.

1.3 Related ideas

The evaluation mechanism used in our system has its origins in theoretical work [19-20] on "call-by-need" parameter passing. The same work has inspired "lazy evaluators" for LISP [21,22] which in some respects behave like our process networks and can execute slightly modified versions of some of our examples. These systems, however, do not have any analogue of our cyclic network structures.

Communication channels are related to the *streams* of Landin, [23] who foresaw their connection with coroutines. In fact, our language can be viewed as a powerful strear processing language. A simplified version of streams already exists in POP-2 [24] in the form of *dynamic lists*, but their usefulness is limited because the lack of processes in POP-2 makes it awkward to define stream transformations.

2. THE PROGRAMMING LANGUAGE

2.1 Introduction

The language presented here provides concise and flexible means for creating complex networks of processes which may evolve during execution. The key concepts are *processes* and structures called *channels* which interconnect processes and buffer their communications. Channels carry information in one direction only from a *producer process* to one or more *consumer processes*, and they behave like unbounded FIFO queues.

In this section we explain how processes are declared, how they communicate via channels, and how networks of processes are created and transformed. Then we introduce a more powerful functional notation and discuss iterative versus recursive reconfiguration of processes.

The language has been implemented in Edinburgh as an extension of POP-2. [20] Although its concrete syntax follows the style of POP-2 (e.g., the Algol assignment "A:=B" is written B → A in POP-2), no feature of POP-2 that departs significantly from PASCAL or ALCOL is used.

2.2 Processes

Each process is specified in a *process declaration*, patterned after a procedure/function declaration in POP-2 :

Process <name> <parameter-list> ;
<process body>
Endprocess

The parameter list is partitioned into two sublists : the *ordinary* parameters and the *port* parameters. Parameters in the first group are evaluated according to the rules of POP-2. Port parameters [25] will be bound to communication channels so that a process may communicate with its neighbors in the network. In the declaration of TRANSDUCER (see fig.1) A is an ordinary parameter, called by value, while QI and QO are respectively input and output ports. In a typed language, say PASCAL, the process heading would appear like this :

Process TRANSDUCER (A: *integer*; QI: *in integer*;
QO: *out integer*);

The body of a process declaration is similar to the body of a procedure in POP-2 : variables and functions may be declared local to the process (thus no restriction is placed on the amount of memory available to a process). Conceptually, each process is thought of as executing on a separate machine so that it cannot communicate with any other process except through interconnecting channels. Hence :

(1) No global variable may be *updated* by a process
(2) Arrays, but *not* references to arrays, may be sent along communications lines
(3) More generally, if a reference to the dynamic storage area is sent by a process, the area that may be reached through this reference must become read-only.

(Note that (3) is automatically satisfied in purely applicative languages). The constraints above do not mean that two processes cannot access a common file or table, but a process should be responsible for the management of all accesses to such a shared object, in the manner advocated in [26]. This is the way, for example, in which we deal with input/output in our system.
All constructs of POP-2 may occur in the body of a process. Two *primitive functions* are provided to transmit information between processes, and a *reconfiguration instruction* allows the redefinition of the network.

2.3 Transmission primitives

A process is connected to its neighbours via one-way communication channels. The function GET is used to obtain data : if A is an input port, the evaluation of the *expression* GET(A) yields the next item arriving via port A. If no value ever arrives, this evaluation does not terminate. Consecutive evaluations of GET(A) yield consecutive items, as if A was a sequential input file. The *procedure call* PUT(<expression>,B) sends the result of evaluation of the first argument along the channel bound to output port B.
Availability of an item at an input port *cannot* be tested. This is not an oversight but a deliberate decision to exclude time-dependent input/output behavior. Certain sections of an operating system *demand* a primitive of this kind, but the absence of time-dependencies gives a distinct flavour to pipelining as opposed to other kinds of parallel processing and permits the simulation of a pipeline by a set of coroutines. Note also that including such a primitive changes drastically the mathematical semantics of the language. [27]

2.4 Reconfiguration

While a process program is running, it may be visualised as a directed graph where nodes represent processes and edges stand for communication channels. During computation, this graph may evolve in a top-down fashion : a node may be replaced by a subgraph, provided this subgraph can be appropriately *spliced* into the incoming/outgoing edges (i.e., channels) of the original node. A *reconfiguration instruction* has the form :

doco <body> *closeco*

and its body specifies a transformation of this sort. The keyword *doco* stands for "do concurrently" or "do coroutines." *Closeco* is just the matching closing bracket. The body of the instruction has two parts :

(i) the declaration of new communication's lines (edges in the new subgraph)
(ii) a list of *process calls*. Port parameters in these calls may be bound either to channels that have just been declared or to ports of the *parent* process, i.e., the process in which this reconfiguration instruction occurs.

In fig.1, the process GO contains a reconfiguration instruction. Its evaluation provokes the graph transformation displayed on fig.2.
The two calls to TRANSDUCER set up two *distinct* instances of this process. As a reconfiguration instruction merely *specifies* a new setup, the order in which process calls occur is, for the moment, irrelevant.*

```
Process PRODUCER out QO;
    vars N;  ∅ → N;
    repeat INCREMENT N;  PUT(N,QO) forever
Endprocess;
Process TRANSDUCER A in QI out QO;
    repeat PUT(A + GET(QI),QO) forever
Endprocess;
Process  CONSUMER in QI;
    repeat 20 times  PRINT(GET(QI)) close
Endprocess;
Process GO;
    doco channels QI Q2 Q3;
    PRODUCER (Q1);  TRANSDUCER(1,Q1,Q2);
    TRANSDUCER(-1,Q2,Q3);  CONSUMER(Q3);
    closeco
Endprocess;

Start doco GO( ) closeco;
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
:

Fig.1. First example.

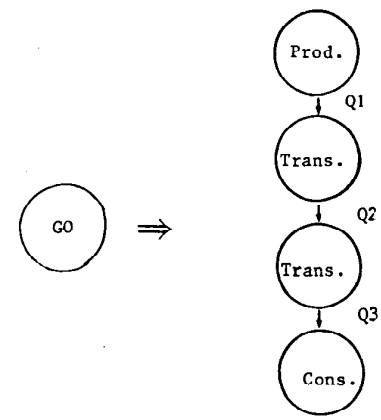

Fig.2. Reconfiguration.

* As the reader realizes by now, the program of fig.1 does not achieve very much. It just serves to illustrate the first features of the language.

Remark : In Algol, a procedure call lumps together three distinct operations : the creation of new procedures, the binding of formal to actual parameters and control transfer. In ISWIM [28] or SL5[29] some of these actions may be performed separately. Here, processes are bound to their arguments as soon as they are created, but control is transferred in an entirely separate manner.
Conway's original coroutine scheme requires the coroutine network to be acyclic. The theory tells us that this restriction is unnecessary. From a pragmatic point of view, the extra cost incurred at execution time is minimal and feedback loops in coroutine programs are *in fact* quite useful. Another constraint, implicit in Conway's paper, is essential. Call a process a *producer* (resp. *consumer*) for Q if it is bound to channel Q via an output port (resp. input port). The sequence of items that will be sent on Q during execution of the program is the *history* of Q. If the history of every communication variable is determined by the value of the program's parameters, a parallel program is called *determinate*. A simple rule guarantees determinacy of process programs : at any given time during execution, a channel must be bound to a single producer, and to a single output port of that producer. Note that several consumers may share the same input line : in this case all consumers get the same input sequence.

2.5 Activation

Within a POP-2 program, an activation instruction of the form :

start <reconfiguration instruction>

may be issued to request execution of a process program. This instruction terminates when the process program terminates. Once an initial network has been set up by the reconfiguration instruction, it is set in motion and from then on control transfer is automatic. In the simple program of fig.1, an activation instruction is issued at top-level. The occurrence of the POP-2 prompt character (':') after the results shows that the program did terminate.

The program of fig.3 is more interesting. This form of the sieve of Eratosthenes appears, to the best of the authors' knowledge, for the first time in [30]. For each newly discovered prime, a FILTER process is created by SIFT, whose task it is to remove all multiples of that prime from the integers to be considered. A proof of the correctness of this program is sketched in section 4.

```
Process INTEGERS out QO;
    Vars N; 1 → N;
    repeat INCREMENT N; PUT(N,QO) forever
Endprocess;

Process FILTER PRIME in QI out QO;
    Vars N;
    repeat GET(QI) → N;
           if (N MOD PRIME)≠ ∅ then PUT(N,QO) close
    forever
Endprocess;

Process SIFT in QI out QO;
    Vars PRIME; GET(QI) → PRIME;
    PUT (PRIME,QO); comment emit a discovered prime;
    doco channels Q;
     FILTER(PRIME,QI,Q); SIFT(Q,QO)
    closeco
Endprocess;

Process OUTPUT in QI; Comment this is a library process;
    repeat PRINT(GET(QI)) forever
Endprocess;

Start doco channels QI Q2;
    INTEGERS(QI); SIFT(QI,Q2); OUTPUT(Q2);
    closeco;
```

Fig.3. Sieve of Eratosthenes.

2.6 Functional notation

The constructs explained so far are sufficient for all programming. However, we can write much more elegant programs in a functional notation. Most processes have a single output line so that they are functions from streams to streams. [23] Thus in the way that ALGOL 60 permits functions along with procedures, processes may be declared functional and used to build stream expressions. In process calls occurring in reconfiguration instructions, such expressions may be provided as arguments where input channels are expected. For example, the program in fig.3 will now look like :

```
Process INTEGERS => QO;
   ...
Process FILTER PRIME in QI => QO;
   ...
Process SIFT in QI => QO;
   ...
Process OUTPUT in QI;
   ...
start doco OUTPUT(SIFT(INTEGERS())) closeco;
```

This new notation is very convenient because many channels are created implicitly. But stream expressions denote only acyclic subnets. A simple construct (akin to Landin's WHEREREC) allows networks to be built with cycles. In a reconfiguration instruction, we allow now a list of *elementary reconfigurations* where, in the familiar BNF notation :

<elem.reconf>::=<process call>

|<elem.reconf>*WHERE*<chan-list>$\{^{IS}_{ARE}\}$<stream-exp-list>

In the program of fig.4, a reconfiguration written in this style can be found in the activation. Since X occurs both in the channel list and within the stream expression on the right of *IS*, a network with a cycle is being specified. Note also that X is shared as input by the three instances of TIMES and the process OUTPUTF.

Finally, a difficulty crops up when, as a consequence of a reconfiguration, the newly created network has to output values on a channel previously bound to an output port of the parent process, as happens in SIFT for example (see fig.3). Another kind of elementary reconfiguration, called *splicing* must be included :

<list of stream expressions> => <list of output ports>

This specifies that the stream expressions on the left are to provide data to the channels bound to the ports on the right. So the reconfiguration in SIFT is now :

```
doco SIFT(FILTER(PRIME,QI)) => QO closeco
```

Note that this device offers a simple way to have a network shrink rather than expand. For example the process QCONS :

```
Process QCONS A in QI => QO;
        PUT(A,QO);
        doco QI => QO closeco
Endprocess
```

first emits A and then ties its input to its output channel and vanishes.

2.7 Optimizing recursion

The situation of the process SIFT is a common one. This process reconfigures into a subgraph containing a new instance of SIFT and disappears. Instead, SIFT could merely create in front of itself new FILTERs upon receiving new inputs and thus be iterative rather than recursive. To indicate that the parent process is to be included in a new configuration, the dummy process call *CONTINUE* is used. Usually the bindings of the parent process have to change. An assignment permits switching inputs, splicing is needed to reconnect outputs. The transformed version of SIFT is :

```
Process SIFT in QI => QO;
  Vars PRIME;
  repeat
    GET(QI) → PRIME; PUT(PRIME,QO)
    doco FILTER(PRIME,QI)→QI; CONTINUE closeco
  forever
Endprocess;
```

In our implementation, significant time and space savings result from this transform.

2.8 An example

The programming style is now much less imperative. To illustrate this, consider a problem treated by Dijkstra. [31] One is requested to generate the first N elements of the sequence of integers of the form $2^a3^b5^c$(a,b,c≥0) in increasing order, without omission or repetition. The idea of the solution is to think of that sequence as a single object and to notice that if we multiply it by 2, 3 or 5, we obtain subsequences. The program of fig.4 embodies the idea that the solution sequence is the least sequence containing 1 and satisfying that property. The process MERGE assumes two increasing sequences of integers as input and merges them, eliminating duplications on the fly. The process TIMES multiplies all elements of its input channel by the scalar A. The process OUTPUTF is a library process. Notice that control considerations do not intervene in the design of this program.

Remarks : (1) What is an implicit quantity in other coroutine systems, the history of a communication's variable ("mytical" variable in [16]), is now explicit and subject to calculations. The style of programming also recalls LUCID, [34] which has a similar semantics. The pay-off will be in easier correctness proofs. Note also that this programming language is just what is needed to compute over real numbers with unlimited accuracy. [35]

```
Process MERGE in QI1 QI2 => QO;
    Vars I1, I2; comment local buffers;
    GET(QI1)→I1; GET(QI2)→I2; Comment initialisation;
    loopif I1<I2 then PUT(I1,QO); GET(QI1)→I1
    elseif I1>I2 then PUT(I2,QO); GET(QI2)→I2
    else comment I1 = I2. Remove duplications;
      PUT(I1,QO);GET(QI1)→I1; GET(QI2)→I2
    close
Endprocess;

Process TIMES A in QI => QO
    repeat PUT(A*GET(QI),QO) forever
Endprocess;

Start doco
      OUTPUTF(20,X)
      where channels X is
            QCONS(1,MERGE(TIMES(2,X),
                          MERGE(TIMES(3,X),
                                TIMES(5,X))))
      closeco
```

1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30 32 36 :

Fig.4. An example from Dijkstra.

(2) Returning to the particular program of fig.4, it can be made much more efficient if we eliminate redundant number generation, calling it with :

```
start doco OUTPUTF(20,X)
      where channels X,Y,Z are
            QCONS(1,MERGE(TIMES(2,X),Y)),
            QCONS(3,MERGE(TIMES(3,Y),Z)),
            QCONS(5,TIMES(5,Z))
      closeco
```

3. EXECUTION

3.1 Outline

From an operational point of view, a process network is a collection of independent machines which interact by making demands upon or sending data along communication channels. Processes are represented by data structures containing local access environments and control continuations. A channel is represented by a linear list containing items stored in the channel and terminating with a reference to the current producer for the channel. Consumers have pointers into this list which are updated by the GET operation, while the producer inserts new items at the end of the list via the PUT operation. A reconfiguration instruction results in :

(1) creation of new processes and channels,
(2) initializing or updating input channel pointers,
(3) initializing or updating channel producer information.

A single constraint regulates the activity of processes : if a process requests input data from an empty channel, it must stop and wait until that data is provided by the channel's producer, which must be activated if possible. Given this constraint, a range of scheduling strategies are possible, from *pure coroutine* execution where a single process is active at any time to *full parallelism* where all processes run except when they are waiting for input. These scheduling strategies all yield the same input/output behavior, because the exclusive use of channels for interprocess communication and the careful choice of data transmission primitives serve to insulate processes from scheduling-dependent information.

3.2 Coroutine mode of execution

In this mode, activation of processes is strictly *demand driven*. Since the demand must originate somewhere, a process is selected to drive the whole network, and the demands of this driving process propagate through the network via the execution of transmission primitives and reconfiguration instructions.

(i) Selection of the driving process

The last process created in the execution of the activation instruction is designated as the initial driving process and is the first process activated. Normally, it is that process which is responsible for producing (e.g. printing) the ultimate outcome.*

(ii) Transmission primitives

Both GET and PUT *may* involve transfer of control. Applying GET to an empty input channel C causes suspension of the running process and activation of the producer for C. The channel C is made *hungry* to indicate that there is a consumer waiting on it. Applying PUT to a hungry output channel causes suspension of the running process and resumption of the waiting consumer, whereupon the interrupted GET operation is completed.

Remarks : (1) There is no transfer of control when GET is applied to a nonempty input channel or PUT is applied to a nonhungry output channel.

(2) If as a result of a GET operation the scheduler attempts to activate a producer which is itself waiting, further computation is impossible and deadlock has been detected.

(iii) Reconfiguration

Except for the driving process, any process which is active is trying to satisfy some hungry output channel. After such a process reconfigures, the scheduler gives control to the (possibly new) producer for that hungry channel.

When the driving process reconfigures it may survive (i.e., remain in the new configuration) in which case it retains control, or it may disappear, in which case the last process created is chosen as the new driving process and is activated.

3.3 Parallel mode of execution

After a PUT instruction sends an item on a hungry channel, the waiting consumer must be reactivated. But if additional processors are available there is

* The last process created in a reconfiguration is the outermost one in the last elementary reconfiguration.

no need to deactivate the producer process ; it may continue to run in *anticipation* of further demands for its output. In this way, computations that were interleaved in time can be made to overlap, and some process switching overhead is saved as well, without increasing the programmer's burden. The one drawback is that the process which was not deactivated *may* carry out *nonessential* computation, i.e., computation that is not needed to produce the final outcome of the program. This nonessential computation may even involve the recursive creation of superfluous processes.
We have developed a method of restraining such over-anticipation and verified its effectiveness in quasi-parallel simulations. The idea is to associate with each channel C a non-negative integer A(C), called the anticipation coefficient. Once activated, the producer for C will not be deactivated by PUT until C contains at least A(C) many unconsumed items. Note that the coroutine mode of execution results when A(C) = 0 for all channels C. The anticipation coefficient can only be accessed by the primitives GET and PUT. It is set by a special primitive ANT(n,C).Normally, the anticipation coefficient should be set as the channel is passed as an input parameter to a new process. In any case, the specifications for anticipation will not affect the semantics of the program, nor will they require alterations to its basic design.

4. PROGRAM PROOFS

For a detailed presentation of the mathematical semantics of the programming language, the reader is referred to [1]. Roughly, to any program one associates a set of recursive equations. Standard proof techniques can then be used. [19,32] A form of *structural induction* is used repeatedly. Suppose one wishes to prove that a sequence X has property P :
(1) First one proves, usually by induction, that P holds for a sufficiently rich set of *finite* initial segments of X.
(2) Second one proves that P *admits induction*, that is that if it holds for sufficiently many finite initial segments of some sequence, it must hold for the whole sequence as well.
Formal proofs are too long to be given here in detail, so we present only the articulating lemmas.

4.1 Program of fig.4

Lemma 1 (Properties of MERGE) : If L_1 and L_2 are strictly increasing sequences of integers then
(1) MERGE(L_1,L_2) is strictly increasing
(2) As sets, MERGE(L_1,L_2) $\subseteq L_1 \cup L_2$ and if L_1 and L_2 are infinite, equality holds.
(3) length (MERGE(L_1,L_2))$\geq$min(length(L_1),length(L_2))

Lemma 2 (Properties of TIMES) : if A is a positive integer, L a strictly increasing sequence of integers then :
(1) TIMES(A,L) is a strictly increasing sequence of integers.
(2) Each element of TIMES(A,L) is the product of A by some element of L.
(3) length(TIMES(A,L))=length(L)

Lemma 3 : The variable X denotes a sequence of infinite length.
Lemma 4 : The solution sequence satisfies the recursive equation defining X.
A special case of McCarthy's recursion induction [33] is applicable here and so we conclude from lemmas 3 and 4 that X is the solution sequence. (The proof of the improved version presents no difficulty.)

4.2 Sieve of Eratosthenes (recursive form)

Lemma 1 : INTEGERS() is exactly the increasing sequence of all integers starting with 2.
Lemma 2 : For any integer p and sequence L, the sequence FILTER(p,L) is a subsequence of L that contains :
(1) no multiples of p
(2) all members of L that aren't multiples of p.

Lemma 3 : For any sequence L, SIFT(L) is a subsequence of L.
Lemma 4 : If L is an increasing sequence and p occurs in SIFT(L) no other multiple of p occurs in SIFT(L).
Lemma 5 : If every element of L is greater than 1 and if p is a prime occurring in L, then p occurs in SIFT(L).

By lemmas 1 and 5, the output of the program must contain all primes. By lemma 4, composite numbers cannot occur. Hence the program generates exactly the primes, in increasing order by lemma 3.

Remark : Notice that in contrast to [21-22] our semantics involves recursively defined data as well as recursively defined functions.

CONCLUSION

In the course of developing this language we have written a considerable number of applications programs, including four types of sorting, a formal power series package with 17 series operations, and a pipeline version of the discrete Fourier transform. This experience has confirmed most of our expectations, indicated limitations, and suggested generalizations.
We have found the language conducive to clear, well-structured programming. Programs are conceived *functionnally* and operational concerns such as process scheduling do not enter into their design. The reconfiguration statement encourages top-down development, and the functional notation provides a concise way of expressing the relationships between processes. Program proofs can be carried out at a level of abstraction which avoids the intricacies of dynamic behavior -i.e., in terms of operations on abstract data rather that machine state transitions.
As our ideas about process networks evolved, channels, considered as data structures, assumed a more central role. A channel is an example of a *dynamic data structure*, i.e., a structure which is gradually generated by processes embedded within itself. To a consumer, these structures behave *as though they were already fully defined*, because as soon as one accesses a new part of the structure it *becomes defined*. Our experience, together with theoretical work by Kahn and Plotkin, suggests that process networks should be generalized by broadening the class of dynamic data structures used for process communication - from linear lists to trees, tableaux, etc. For example, in a compiler the abstract syntax tree might be a dynamic tree generated from the input text by a number of parser processes operating in parallel, while several consumer processes work from the top down generating code.
As a final comment, this study seems to provide further evidence for developing the model theory of programming languages. [36] As the level of expression in programming languages increases, their interpreters will also become increasingly sophisticated. In the design and proof of programs, a firm grip on the model theory of the language will prove more useful than the knowledge of a delicate (and mythical) interpretive mechanism.

ACKNOWLEDGMENT

We owe thanks to Rod Burstall, Gordon Plotkin, and Jerry Schwarz for many helpful discussions. The authors gratefully acknowledge support from (respectively) the Compagnie Internationale des Services en Informatique and the U.K. Science Research Council.

REFERENCES

[1] Gilles Kahn, The semantics of a simple language for parallel programming, Proceedings of IFIP CONGRESS 74, North-Holland Publ. Co, 1974.

[2] Melvin E. Conway, Design of a separable transition-diagram compiler, Communications of the ACM, vol. 6, no 7, July 1963, 396-408.

[3] Richard M. Karp and Raymond E. Miller, Parallel program schemata, Journal of Computer and System Sciences, vol. 3, no 1, 1969, 147-195.

[4] Duane Adams, A computation model with data flow sequencing, Ph.D. Dissertation, Stanford University, Computer Science Dept., December 1969.

[5] Jack B. Dennis, On the design and specification of a common base language, in Computers and Automata, Brooklyn Polytechnic Institute, 1971.

[6] Donald E. Knuth, The art of computer programming, Fundamental Algorithms, vol. 1, Addison-Wesley, Reading, Mass. 1968.

[7] Brian W. Kernighan and P.J. Plauger, Software Tools, Addison-Wesley, Reading, Mass. 1976.

[8] D. Barton, I.M. Willers and R.V.M. Zahar, An implementation of the Taylor series method for ordinary differential equations, The Computer Journal vol. 14, no.3, 243-248.

[9] Donald E. Knuth, The art of computer programming Sorting and searching, vol. 3, Addison-Wesley, Reading, Mass. 1973.

[10] W.K. Pratt, J. Kane and H. Andrews, Hadamard transform image coding, Proceedings of the IEEE, vol. 57, no.1, 58-67.

[11] J.L. Stansfield, Programming a dialogue teaching situation, Ph.D. Thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1974.

[12] Dennis M. Ritchie and K. Thompson, The UNIX operating system, Communications of the ACM, vol. 17, no. 7, July 1974.

[13] Ole-Johan Dahl and C.A.R. Hoare, Hierarchical program structures, in Structured Programming, Academic Press, 1972.

[14] W. Morven Gentleman, A portable coroutine system, Proceedings of the IFIP CONGRESS 1971, North-Holland, 1971.

[15] G. Lindstrom, Control extension in a recursive language, BIT, vol. 13, no.1, 1973, 50-70.

[16] Maurice Clint, Program proving : coroutines, Acta Informatica, vol. 2, no.1, 1973, 50-63.

[17] P.A. Pritchard, A proof rule for multiple coroutine systems, Information Processing Letters, vol. 4, no.6, March 1976.

[18] Ole-Johan Dahl, An approach to correctness proofs of semi-coroutines, Symposium on Mathematical Foundations of Computer Science, A. Blikle Ed., Springer Verlag, 1976, 157-174.

[19] Jean E. Vuillemin, Proof techniques for recursive programs, Ph.D. Thesis, Stanford University 1973.

[20] Christopher P. Wadsworth, Semantics and pragmatics of the lambda-calculus, Ph.D. Thesis, University of Oxford, September 1971.

[21] James H. Morris and P. Henderson, A lazy evaluator, Proceedings of the Third ACM Conference on Principles of Programming Languages, January 1976.

[22] D.P. Friedman and D.S. Wise, CONS should not evaluate its arguments, Third International Colloquium on Automata, Languages and Programming, Edinburgh University Press, 1976.

[23] Peter J. Landin, The correspondence between ALGOL 60 and Church's lambda notation : Part 1, Communications of the ACM, vol. 8, no.2, February 1965, 89-101.

[24] R.M. Burstall, J.S. Collins and R.J. Popplestone, Programming in POP-2, Edinburgh University Press, 1971.

[25] Robert M. Balzer, An overview of the ISPL computer system design, Communications of the ACM, vol. 16, no.2, February 1973, 117-122.

[26] Carl Hewitt, et al. Behavioral semantics of nonrecursive control structures, Colloque sur la Programmation, Springer Verlag, 1976.

[27] Gordon Plotkin, A powerdomain construction, to appear in SIAM Journal on Computing.

[28] Peter J. Landin, A lambda-calculus approach, in Advances in Programming and Non-numerical Computation, Pergamon Press, 1966.

[29] D.E. Britton, F.C. Druseikis, R.E. Griswold, D.R. Hanson and R.A. Holmes, Procedure referencing environments in SL5, Third ACM Symposium on Principles of Programming Languages, January 1976.

[30] M. Douglas McIlroy, Coroutines, Internal report, Bell Telephone Laboratories, Murray Hill, New Jersey, May 1968.

[31] Edsger W. Dijkstra, A discipline of programming, Prentice Hall, New Jersey, 1976.

[32] Robin Milner, Implementation and applications of Scott's logic for computable functions, ACM Conference on Proving Assertions about Programs, January 1972.

[33] John McCarthy, A basis of a mathematical theory of computation, in Computer Programming and Formal Systems, Braffort and Hirschberg, Ed., North-Holland, Amsterdam, 1963.

[34] Edward A. Ashcroft, William Wadge, Proving programs without tears, Symposium on Proving and Improving programs, IRIA, Rocquencourt, G. Huet and G. Kahn Ed., 1975.

[35] Edwin Wiedmer, Exaktes rechnen mit rellen Zahlen, Eidgenössische Technische Hochschule, Zürich, Bericht no. 20, July 1976.

[36] Dana Scott, Outline of a mathematical theory of computation, Proceedings of the Fourth annual Princeton Conference on Information Sciences and Systems, 1970, 169-176.