SNPD '00, University of Reims, France, May 2000

First Steps in Computer Science: Sequential or Parallel?

Shortened from "Simple Flexible Language - SFL" [13],
which is available at "http://shekel.jct.ac.il/~rafi".

R.B. Yehezkael (formerly Haskell).
Jerusalem College of Technology - Machon Lev,
Hawaad Haleumi 21, Jerusalem 91160, ISRAEL.
Tel: 02-6751111 e-mail: rafi@mail.jct.ac.il

(Minor corrections March 2001 - אדר תשס״א.)

Abstract

In educating computer scientists, typically the first steps deal with sequential algorithms and programs. This has unfortunate consequences in that the students mind becomes geared to a sequential way of thinking. Parallel programming on the other hand is very hard so it is not practical to introduce these concepts at an early stage. Our approach is to define a simple flexible language (SFL) which does not obligate sequential or parallel execution but enables these execution methods.

This language has several faces. It can be viewed as a programming language, capable of being executed in a variety of ways, sequential or parallel, with identical end results. It can also be viewed as a command language for the user. Also, hardware block diagrams can be derived from "programs" in this language.

Mathematical proofs using specifications and computational induction are clear and straightforward. It is a good choice for teaching computer science, and computer systems engineering concepts in view of its ability to describe both software and hardware aspects.

The "core" language is based on programs (procedures or functions) with "IN", "OUT" but no "INOUT" parameters, and conditional statements. As in mathematics, variables, parameters, etc., receive a value once only. Blocks, loops, case statements etc. (including nested forms) are not part of the "core" language but viewed as abbreviations for certain compound forms in the "core" language.

1. Introduction

Conventional programs can change the values of variables during the course of execution, and so to ensure uniquely defined results, sequential execution is required. The advantage of the sequential approach is that it is easy to implement and easy to follow the execution steps. The analysis and debugging of these programs is hard in view of the fact that program statements are not that independent of each other and so many interactions need to be considered; and a particular value can depend on a change made many operations in the (distant) past. Parallel programming is even harder, particularly if the programmer explicitly handles the coordination.

Our approach is to define a simple flexible language in which there is greater independence between the statements of the language. Though writing programs in such a language is harder than writing conventional programs, the fact that there is more independence between the statements of the language has advantages.

1. Programs may be executed in a variety of orders, sequential and parallel, with identical end results.
2. The analysis and debugging of the programs is easier, in view of the fact that greater independence between program statements means there are fewer interactions to be considered.

A lesson from mathematics: The subtlety of the simple mathematical variable is that it allows computations to be performed in a variety of orders sequential and parallel. It

also enables only some of the variables to be computed. So partial computation is possible. One of the reasons for the flexibility is that variables may not be updated. Another reason for this flexibility is that a simple mathematical variable is really a function of zero arguments in programming language terms [13].

The SFL approach: In SFL we take an approach which is in-between the mathematical view and the procedural programming view.

1. Like mathematical variables, SFL variables receive a value once only, but they are not functions of zero arguments.
2. Like mathematical notation, computation may proceed in a variety of orders, sequential and/or parallel.
3. Like the procedural programming approach, we require all steps to be performed. So partial computation is not supported.

2. A Flexible Program

Let us now give an example of a simple program in this flexible language, describe different execution methods, and analyze the program for correctness.

```
function reverse (IN vector v; integer low, high;
                  OUT vector v');
/*
SPECIFICATION:
IN - "v" is a vector and "low", "high" are positions within the vector v.
OUT - If low≤high, then within the range "low" to "high", v' is like v but reversed. Other elements of v' are not given values by this function.
If low>high, then the function does nothing to v'.
*/
{
if (low<high)
   {v'(high) = v(low);
    v' = reverse (v, low+1, high-1); /* A*/
    v'(low) = v(high);}
else if (low==high)
        {v'(high) = v (high);};
} /* end reverse */
```

For example, to reverse a vector (1, 2, 3, 4, 5) and put the result in r' we write:

```
r' = reverse ((1, 2, 3, 4, 5), 0, 4) ; /* B */
```

Assumptions:

1. The first position or subscript in a vector is zero.
2. Later on /* A */, /* B */ will be used as return addresses, when describing execution methods.
3. Assignment is denoted by "=" and equality by "==".

Notes:

1. Parameters may only be IN which are given first or OUT which come last. There may be several IN parameters and several OUT parameters.
2. The tag is only used after the name of OUT parameters. This aids readability.
3. It is an error to assign twice to the same simple parameter or simple component of a parameter having (several) components.

Different ways of writing parameters inside a function call:

The definition of reverse given previously included a call in functional style:

```
v' = reverse (v, low+1, high-1);
```

Sometimes procedural style is clearer in the form:

```
reverse (v, low+1, high-1, v');
```

Sometimes an assignment like style is helpful:

```
reverse (v=v; low=low+1; high=high-1; v'=v');
```

This can be abbreviated showing only the changes:

```
reverse (low=low+1; high=high-1);
```

Note that there is a fundamental difference between low=low+1, high=high-1 written above and the assignment statement which updates values. Here the variables "low", "high", on the left hand side are new variables which will be created when the call is executed and the variables "low", "high" on the right hand side are existing variables holding values i.e. the variables on different sides of the "=" are different variables. Though this may look like we are updating the values of variables, this most definitely is not the case. In fact, we are giving new variables their values. So flexible execution remains possible with this arrangement. (In certain cases these statements may be performed as assignment statements, for example for singly tail recursive, flowchart type, function definitions being executed sequentially - later.)

3. Execution Methods

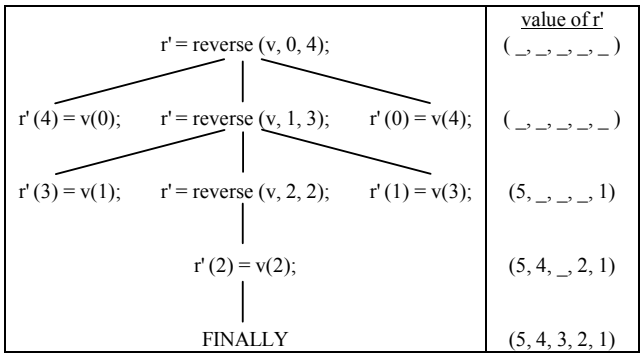
The restrictions described above allow the execution to be performed in a variety of orders with equivalent end results. We present three execution methods.

1. Parallel shown in tree form.
2. Sequential using a stack.
3. Sequential using a queue.

We also discuss how to use a waiting area for efficient virtual memory handling.

In presenting these methods we assume that v = (1, 2, 3, 4, 5) and that we are executing: r' = reverse (v, 0, 4); /* B */

Parallel execution shown in tree form:



Note that though the values of r' are determined assuming that the computation proceeds level by level (synchronously), the only execution order requirement is that if there is a line joining two instructions, the higher is executed before the lower.

The above can also be represented by a sequence of pairs of the form set of expressions::values of OUT variables as follows:

{ r' = reverse (v, 0, 4); }::r'=(_, _, _, _, _)
≡{ r' (4) = v(0); r' = reverse (v, 1, 3); r' (0) = v(4); }
:: r'=(_, _, _, _, _)
≡{ r' (3) = v(1); r' = reverse (v, 2, 2); r' (1) = v(3); }
:: r'=(5, _, _, _, 1)
≡{ r' = reverse (v, 2, 2); }:: r'=(5, 4, _, 2, 1)
≡{ r' (2) = v(2); }:: r'=(5, 4, _, 2, 1)
≡{ }:: r'=(5, 4, 3, 2, 1)

Sequential execution using a stack:

Here calls are executed immediately and a record is made in the stack of the value of variables at the time of the call. The return address is also recorded in the stack. (This is the standard way of handling calls in procedural programming languages.) Note that the top of the stack is at the right and A, B denote return addresses.

Stack			Value of r' at	call/exit
v, 0, 4, r', B			(_, _, _, _, _)	call
v, 0, 4, r', B	v, 1, 3, r', A		(_, _, _, _, 1)	call
v, 0, 4, r', B	v, 1, 3, r', A	v' 2, 2, r', A	(_, _, _, 2, 1)	call
v, 0, 4, r', B	v, 1, 3, r', A	v' 2, 2, r', A	(_, _, 3, 2, 1)	exit
v, 0, 4, r', B	v, 1, 3, r', A		(_, 4, 3, 2, 1)	exit
v, 0, 4, r', B			(5, 4, 3, 2, 1)	exit

Sequential execution using a queue:

Here the calls are delayed and stored in a queue until the current function completes execution. Note that the head of the queue is at the left.

queue		value of r' at exit
r' = reverse (v, 0, 4)		
r' = reverse (v, 0, 4)	r' = reverse (v, 1, 3)	(5, _, _, _, 1)
r' = reverse (v, 1, 3)		
r' = reverse (v, 1, 3)	r' = reverse (v, 2, 2)	(5, 4, _, 2, 1)
r' = reverse (v, 2, 2)		(5, 4, 3, 2, 1)

Using a waiting area for efficient virtual memory handling:

By adding a waiting area we can improve virtual memory performance of the previous three methods. Let us illustrate this assuming that sequential execution using a queue is used.

Let us assume that if an operand of an assignment (or any primitive operation) is not in physical memory, then the hardware (or operating system) will add that operation to a waiting area and continue executing the program. Execution of the instructions in the waiting area may take place in a variety of ways:

1. When the waiting area is (sufficiently) full
2. When the queue is empty
3. At periodic intervals.
4. Combinations of the foregoing.

What is important concerning SFL is that the instructions in the waiting area can often be rearranged, as once only assignment and parameters of type IN or OUT only, allow various execution orders. So the operating system can arrange the order so as to minimize disk access time (more explanation in [13]).

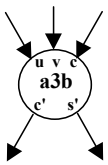
4. Hardware Block Diagrams

Here is an example regarding addition of bit vectors, where we assume there is hardware operation "a3b" for

adding three bits giving their carry and sum respectively (i.e. a full adder). Here is a specification of "a3b".

```
function a3b (     IN bit u, v, c;
            OUT bit c', s';
/*
SPECIFICATION:
IN - u, v, c, are the bits to be added.
OUT - c' is the carry and s' is the sum.
*/
{    /* This is a hardware operation */
}; /*a3b*/
```

This operation can be represented diagrammatically as follows:



Here is an SFL program for adding two bit vectors with a (previous) carry bit.

```
function add ( IN bitvector u, v; bit c; integer n;
                    OUT bit c'; bitvector s' );
/*
SPECIFICATION
IN - "n" denotes the position of the least significant bits of u,v, to be added, where n≥0. The position of the most significant bit "0". Bit "c" is also added at the position of the least significant bits of u, v.
OUT - the carry of the addition is produced in c' and the sum itself in s'.
*/
{
  if (n>0)
  add( c, s'(n)= a3b(u(n), v(n), c); n=n-1);
  else c', s'(0)=a3b(u(0), v(0), c);
} /* add */
```

NOTE: In the above program, the occurences of c on opposite sides of the "=" denote different variables and similarly for n.

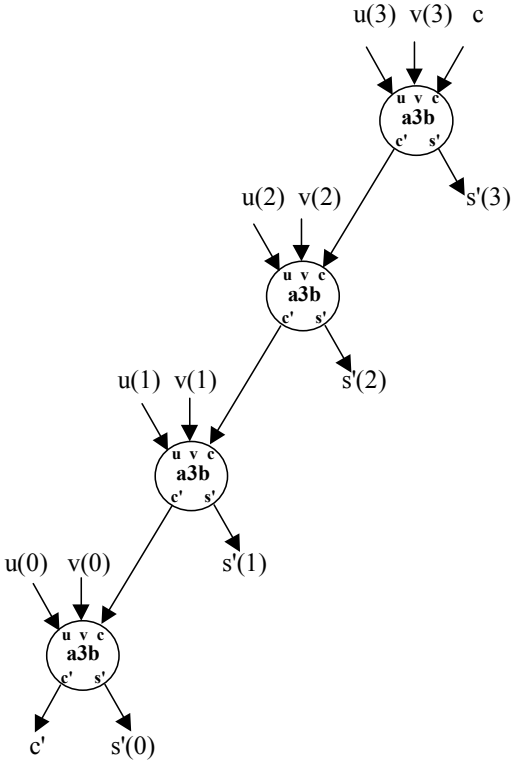
Let us now use symbolic execution on the set { c', s'=add(u, v, c, 3); } so as to get a block diagram of a circuit for adding bits in the range 0 to 3, that is a 4 bit adder. The execution is shown as a sequence of equivalent sets of statements as follows:

```
{c',s'=add(u,v,c,3);}
≡{add(c,s'(3)=a3b(u(3),v(3),c);n=2);}
≡{add(c,s'(2)=a3b(u(2),v(2);
          c,s'(3)=a3b(u(3),v(3),c));n=1);}
≡{add(c,s'(1)=a3b(u(1),v(1);
          c,s'(2)=a3b(u(2),v(2);
             c,s'(3)=a3b(u(3),v(3),c)));n=0);}
≡{c',s'(0)=a3b(u(0),v(0);
     c,s'(1)=a3b(u(1),v(1);
        c,s'(2)=a3b(u(2),v(2);
           c,s'(3)=a3b(u(3),v(3),c))));}
```

Perhaps this is clearer if we explicitly show the "IN" formal parameter names of "a3b" in the following way.

```
≡{c',s'(0)=a3b(u=u(0); v=v(0);
     c,s'(1)=a3b(u=u(1); v=v(1);
        c,s'(2)=a3b(u=u(2); v=v(2);
           c,s'(3)=a3b(u=u(3); v=v(3); c=c))));}
```

The last set contains only hardware operations. It essentially describes the following full block diagram for carrying out the add function. Note that all occurrences of the single bit variables u, v, c denote different variables and is seen clearly from the diagram:



The textual presentation of symbolic execution is difficult to follow in view of the fact that there are several variables all with the same name! A diagrammatic approach, though longer, can make things clearer. The diagrammatic description of the add function and the derivation of the previous block diagram using symbolic execution in diagrammatic form, is shown in [13].

5. Teaching These Concepts

Here is a suggested syllabus for a course which presents the concepts. We propose that significant time should be devoted to the reading and analysis of SFL programs before writing SFL programs.

Title:

Introduction to methods of computation

Objectives:

To provide the student with an overview of various methods of computation.

To broaden the students ability in handling various methods of computation through the use of a simple flexible language.

Topics:

Summary
Introduction
Simple programs
Execution methods
Program verification
Simple program transformations
Hardware block diagrams
Skeleton programs and data flow diagrams
Architecture diagrams
Flowcharts and SFL
Tail recursion with single and multiple tails
Modeling standard programming language constructs
The user interface
Conclusion

Exercises:

Reading and student execution of programs.

Reading and verifying programs with respect to their specifications.

Completing skeleton programs.

Writing skeleton programs and complete programs from specifications and their verification.

Writing specifications, skeleton programs and complete programs and their verification.

Program design - skeleton programs and data flow diagrams.

6. Other Topics

Because of space limitations we do not discuss the following topics here (see [13]).

1. Program verification.
2. Data flow diagrams
3. Architecture diagrams.
4. Flowcharts and SFL.
5. Convenient extensions.
6. The user interface

7. Conclusions

- Many faces to the language
- Once only assignment contributes to greater independence between program statements.
- IN or OUT parameters only help give clarity to the program
- Writing a program is expected to be harder than in a sequential programming language.
- Program analysis and debugging is expected to be easier.
- Hardware block diagrams, Data flow diagrams, Architecture diagrams can be produced from SFL programs (see [13]). Skeleton SFL programs can be produced from Data flow diagrams (see [13]).
- Flexible execution and implementation of programs.

Comparison with other kinds of languages:

SFL is a low level functional language which has connections with hardware and system architecture. Blocks declarations and conditionals are written in the style of C/Java [1, 2]. IN and OUT parameters, vectors are written in the style of ADA [3]. It differs from conventional algorithmic languages in that variables are either IN or OUT and assignment is once only.

It differs from other functional languages [4, 5] in that variables (e.g. a vector) may hold unassigned (unknown) elements. If we were to write a program for reversing a

vector in other functional languages, at each swap a new vector would be created causing gross inefficiency. In SFL, only one new vector was created. It also differs from other functional languages in that they are higher level languages which make more use of higher order functions. SFL on the other hand is more algorithmic in style.

SFL is closest to an early version of LUCID though today LUCID is a data flow language [6]. In LUCID, a well formed program cannot cause a multiple assignment, i.e. it is a compile time check, but in SFL it is a run time error. As mentioned above regarding the reverse example, this can give significant execution improvements when processing vectors and other multi-component data. It also allows greater expressiveness. It is less efficient however, for handling single component data.

SFL handles multiple assignment at run time in a way similar to which the "Id-" and "pH" languages [7, 8] handle I-structures. Unlike the "pH" language, SFL does not have mutable structures (M-structures).

There is a significant difference with logic programming languages [9, 10] in that logic programming languages support non-determinism (multiple results). Common features include once only assignment and support for unassigned (unknown) elements. Also, some logic programming languages support IN and OUT parameters.

Configuration languages [11, 12] are used for specifying the interconnections of distributed systems and SFL bears similarities to configuration languages in the way in which variables and assignments are handled. We therefore expect that by adding appropriate data structures, this language can be used like configuration languages, for describing the communication links of distributed systems. However, further work is needed here. In any case, SFL as a programming language is a good companion to configuration languages in view of its flexible execution and implementation possibilities.

To sum up, SFL is a flexible functional language with an algorithmic style, and may be easier for programmers and engineers to use, compared to other functional and logic programming languages. Its flexibility and many faces are important in education and design.

References

[1] "The C Programming Language", B.W. Kernigham and D.M. Ritchie, Prentice Hall 1978

[2] "The Java Language Specification, Version 1.0", J. Gosling, B. Joy and G. Steele, Addison Wesley 1996

[3] "The Annotated Ada Reference Manual", ANSI/MIL-STD-1815A-1983 (annotated), K.A. Nyberg (Editor), Grebyn Corporation 1989

[4] Functional Programming: Languages Tools and Architectures, S. Eisenbach, Imperial College, London, Halsted Press: A division of John Wiley and Sons INC., 1987

[5] "Report on the Programming Language Haskell, A Non-strict Purely Functional Language", Paul Hudak et.al., Yale University Research Report No. YALEU/DCS/RR-777, 1st March 1992.

[6] "LUCID, the Dataflow Programming Language", W.W. Wadge and E.A. Ashcroft, Academic Press, 1988.

[7] "Compilation of Id-: a subset of ID", Z.M. Ariola and Arvind, MIT Computer Structures Group Memo 315, revised 1 November 1990

[8] "pH Language Reference Manual, Version 1.0-preliminary", R.S. Nikhil et. al., MIT Computer Structures Group Memo 369, 31 January 1995

[9] "Programming in Prolog", " W.F. Clocksin, and C.S. Mellish, Springer Verlag, 2nd edition 1984.

[10] "Concurrent Prolog - Collected Papers Vols 1,2" edited by Ehud Shapiro, MIT Press, 1987.

[11] "An Introduction to Distributed Programming in REX", J. Kramer, J. Magee, M. Sloman, N. Dulay, S.C. Cheung, S. Crane, and K. Twiddle, in "Proceedings of Esprit, Brussels, 1991.

[12] "Structuring Parallel and Distributed Programs", J. Magee, N. Dulay, and J. Kramer, in "Proceedings of the International Workshop on Configurable Distributed Systems, London, 1992.

[13] "Simple Flexible Language - SFL", R. B. Yehezkael, Internal paper, Jerusalem College of Technology - Machon Lev, Revised 2000, available at http://shekel.jct.ac.il/~rafi