ב״ה

SIMPLE FLEXIBLE LANGUAGE - SFL

SUMMARY

A simple flexible language (SFL), having several faces, is described. The language 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.

Because of its simplicity, hardware, architecture and data flow diagrams can be derived from "programs" in this language.

Because of its simplicity, 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 simplicity and flexibility to describe both software and hardware architecture.

The "core" language is based on programs (procedures or functions) with "IN", "OUT" but no "INOUT" parameters, and conditional statements, with or without an "ELSE". 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.

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.

NOTES

1) This document presents in an informal way ideas, considerations and principles concerning the design of SFL, a flexible language with several faces. It does not however fix a definitive form for this language.

2) An earlier version of this paper has been presented to the Compiler Group Meeting, IBM Research Laboratories, Haifa, 3rd May 2000.

CONTENTS

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 much more 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.

The notations of classical mathematics and procedural programming languages

Consider of the following:

Mathematical View	Definitions/Statements	Procedural Programming View
Values of variables independent of order of definitions. No concept of previous value of variable. Execution order not explicitly specified.	x = b - c a = c + 2 b = 4 c = 1 y = x / (x + 3) z = b + c	Values of variables depend on order of statements and previous values of variables. Execution sequential.

Notes

1) Though for example, x = x+1 is acceptable as a statement in procedural programming languages its use is unacceptable in mathematics.
2) If all we needed is the value of z, mathematically only b, c, z need to be determined but in procedural programming all steps would be executed.

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. Here is the programming equivalent of the mathematical view in the style of "C/Java".

```
int function x( ) {return b( ) – c( )}
int function a( ) {return c( ) + 2}
int function b( ) {return 4}
int function c( ) {return 1}
int function y( ) {return x( ) / (x( ) + 3)}
int function z( ) {return b( ) + c( )}
```

There are difficulties with using the above definitions with the procedural programming approach. For example, when computing y ( ) we will compute x( ) twice even though the same value will be produced on both occasions. This can be

overcome by using static storage for the function values and for status information about the state of the variable (i.e. unassigned, assigned, in computation, value in error). In this way, we can handle the computation flexibly without unnecessary re-computation. However, even with this improvement functions of zero arguments will be less efficient than variables of procedural programming language.

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.

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; int 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.)

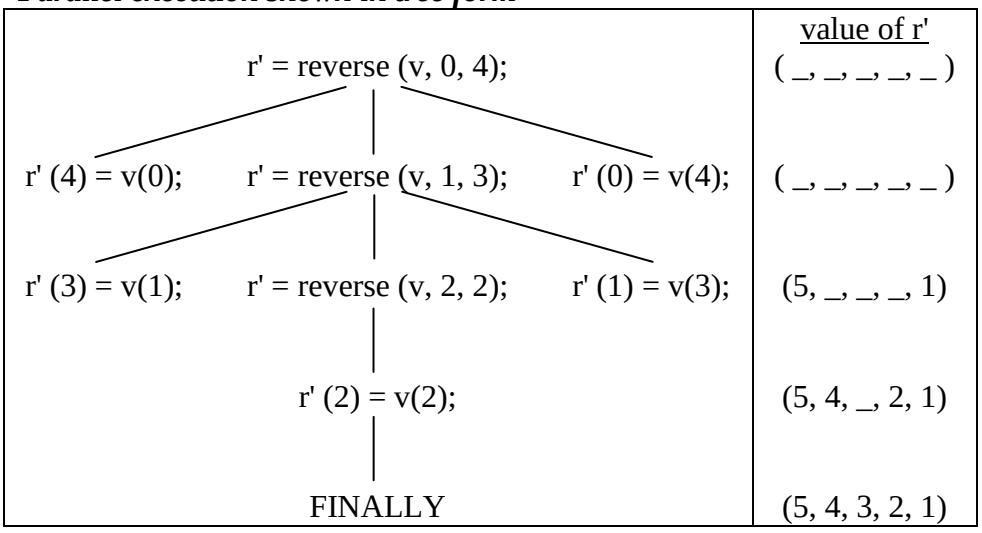
EXECUTION METHODS

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

1. Parallel shown in tree form.
2. Sequential using a stack, with immediate calls.
3. Sequential using a queue, with delayed calls.
4. Sequential using a current function data area, and a stack with delayed calls.

We also discuss how to use a waiting area for efficient virtual memory handling.
In presenting these methods we assume 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 statements::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)

NOTE: Though we have used sets here, the use of a multiset or bag may enable more effective use of parallelism.

Sequential execution using a stack with immediate calls

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: Top of stack is at the right. A, B denote return addresses.

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

Sequential execution using a queue with delayed calls

Here the calls are delayed and stored in a queue until the current function completes execution. (In a composition, an outer call is queued after an inner call.) There are no return addresses in the queue, but a note is made of the function to be activated in the future, and its parameters. When the current function completes, its entry is removed from the head of the queue, and processing continues with the next entry.

Note: Head of 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)

This method does not handle all cases, but will handle tail recursive definitions even if there are multiple tails. The usual definition of tail recursion requires a single tail and is a recursive form which forces sequential execution and is equivalent to flowcharts. The following definition of f is not tail recursive in the usual sense (single tail). It has multiple tails - the calls to g1, g2, g3, which are functions defined by the user (not primitive functions). This kind of definition can be executed sequentially by the queue. (It also enables g1(...), g2(...), g3(...) to be executed in parallel using other execution methods.)

```
function f (IN int i, j;  OUT int m', n', p');
{
if (...)
  {m'=g1(i+1);
   n'=g2(j+1);
   p'=g3(i+j);}
else
   {m'=0; n'=1; p'=2;};
} /* f */
```

Sequential execution using a current function data area, and a stack with delayed calls

Here the data for the current function are stored in a current function data area off the stack, preferably in registers. Calls are delayed and stored in a stack until the current function completes execution. (In a composition, an outer call is stacked before an inner call.) There are no return addresses in the stack, but a note is made of the function to be activated in the future, and its parameters. When the current function completes, the stack is popped into the current function data area, and the function recorded therein started.

Stack	Current function data area	value of r' at exit
	r' = reverse (v, 0, 4)	
r' = reverse (v, 1, 3)	r' = reverse (v, 0, 4)	(5, _, _, _, 1)
	r' = reverse (v, 1, 3)	
r' = reverse (v, 2, 2)	r' = reverse (v, 1, 3)	(5, 4, _, 2, 1)
	r' = reverse (v, 2, 2)	(5, 4, 3, 2, 1)

The execution is similar to that of the queue, but there can be differences in the order of the delayed calls.

This method is restricted in that it can not for example, handle a function in the condition part of an "if" statement. However, by rewriting the "if" even this case can be handled as follows.

```
if (G(...))
statement1;
else statement2;
```

This would be rewritten (by a compiler) as an internal block and a new local variable as follows:

```
:( boolean new_variable=G(...) )     /* Local variable(s) with their initial value. */
{
if (new_variable)
statement1;
else statement2;
}
```

(A block can be removed by using an auxiliary function, as explained in the section "CONVENIENT EXTENSIONS". This results in such an "if" being converted to a function composition.)

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.

Let us assume that v is not in physical memory and r' is in physical memory. Let us assume that the elements of v are in five consecutive disk tracks for example - assume this for explanatory purposes. Then when the queue becomes empty the waiting area would contain five assignments.

1) r' (4) = v(0)
2) r' (0) = v(4)
3) r' (3) = v (1)
4) r' (1) = v (3)
5) r' (2) = v (2)

Based on the disk locations of the elements of v and disk head position, the operating system can execute these instructions in an order which optimizes disk access. In this case, optimal orders are 1, 3, 5, 4, 2 or 2, 4, 5, 3, 1 depending on which side of v the disk head is located. There may be other optimal orders, if the disk head is over an element of v.

PROGRAM VERIFICATION

Computational induction is a technique for proving or justifying that a program is partially correct, i.e. halting is not guaranteed, but whenever the program halts normally, the results are correct according to the specifications. Proof of halting or total correctness will not be discussed here but we can gain confidence that a program halts by thorough testing.

Actually, computational induction is based on a form of simple induction on the length of the computation. Let us now present two parallel proofs that the function

reverse is partially correct using both these techniques. We use two columns where there are differences between the proofs and write the common parts once only across the width of the page.

Computational Induction.	Induction on length of computation.
	If the computation does not halt normally, we have nothing to prove. When the computation halts normally, we use induction on the length of the computation to show that reverse is partially correct.

Originally the elements v' are all undefined, that is:-
....., _ , _ ....., _ , _ ,.....

If low<high we execute
v'(high) = v(low);
v' = reverse (v, low+1, high-1);
v'(low) = v(high);

If we execute only the two assignments v' will look like:-
....., v(high) , _ ....., _ , v(low) ,.....
We have to execute v' = reverse (v, low+1, high-1).

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call reverse(v,low+1,high-1) works correctly.	Since the computation halts normally, the length of computation of reverse(v,low+1,high-1) is shorter than the length of computation of reverse(v,low,high). Therefore by simple induction we may assume that reverse(v,low+1,high-1) works correctly.

So this means that the elements of v' between low+1 and high-1 will be like the elements in the corresponding positions in v but reversed. This means that the elements of v' will now be :-

.....,v(high),v(high-1),.....,v(low+1),v(low),.....

Which means that all the elements of the vector v' between low and high are like the elements of v but reversed.

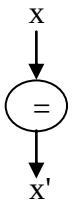
If low==high then v'(high)=v(high) and this clearly conforms to the specification of reverse.

If low>high then the procedure does nothing to v' which is in agreement with the specification.

So the above function is partially correct i.e. halting is not guaranteed, but whenever the program halts normally, the results are correct according to the specifications..

HARDWARE BLOCK DIAGRAMS

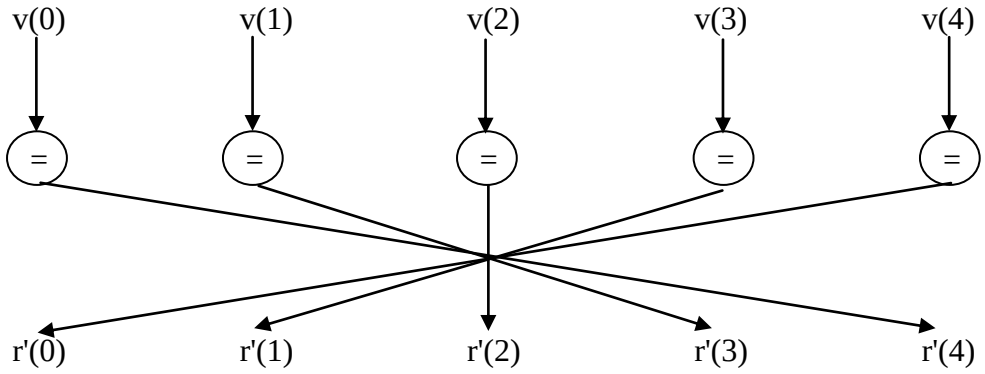
Let us say we have a circuit for performing a copy operation x' = x.



We can produce a block diagram of a circuit for reversing a vector of five elements from the set { r' = reverse (v, 0, 4); } by symbolically executing the program (i.e. substituting using function definitions as in mathematics). The execution is shown as a sequence of equivalent sets of statements as follows:

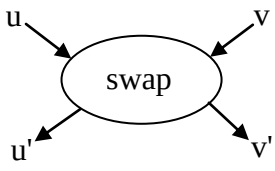
{ r' = reverse (v, 0, 4); }
≡ { r' (4) = v(0); r' = reverse (v, 1, 3); r'(0) = v(4); }
≡ { r' (4) = v(0); r'(3) = v(1); reverse (v, 2, 2); r'(1) = v(3); r'(0) = v(4); }
≡ { r' (4) = v(0); r'(3) = v(1); r' (2) = v(2); r'(1) = v(3); r'(0) = v(4); }

As the last line contains only primitive operations, it essentially describes the following block diagram for carrying out the reverse. (As no output is being generated, there is no need give the state of the output variables, which we gave in previous examples of actual computations.)



So, by a process of symbolic execution, we obtain in this case a block diagram of a circuit for carrying out evaluations when the range to be reversed is fixed.

Let us suppose that the hardware has a swap operation which is equivalent to the function




```
function swap  ( IN int u, v;
                 OUT int u', v');
/* SPECIFICATION: u' = v, v' = u   */
{
u' = v; v' = u;
} /* end swap */
```

We can now rewrite reverse to take advantage of this new operation as follows, where we call the new function "rev".

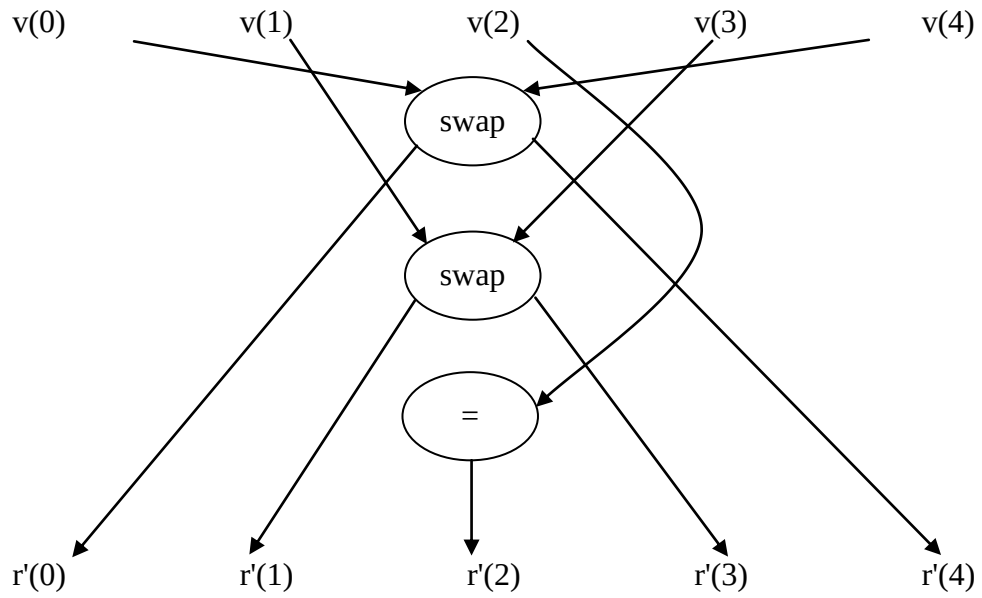
```
function rev  (IN vector v; int low, high;
               OUT vector v');
{
if (high > low)
{ v'(low), v'(high) = swap ( v(low), v(high) );
v' = rev (v, low +1, high -1); }
else if (high==low)
{ v'(high) = v(high)};
} /* end rev */
```

Note: as swap returns two values, the left hand of the assignment has two variables, separated by commas, for receiving these values.

Now we apply symbolic execution to { r' = rev (v, 0, 4) }; to produce a sequence of equivalent sets of statements as follows:

```
{ r' = rev(v, 0, 4) }
≡ { r'(0), r'(4) = swap ( v(0), v(4) );
    r' = rev (v, 1, 3,); }
≡ { r'(0), r'(4) = swap ( v(0), v(4) );
    r' (1), r'(3) = swap ( v(1), v(3) );
    r' = rev (v, 2, 2); }
≡ { r'(0), r'(4) = swap ( v(0), v(4) );
    r'(1), r'(3) = swap ( v(1), v(3) );
    r'(2) = v (2); }
```

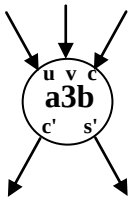
This corresponds to the block diagram.



Here is another 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 by a diagram 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; int 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.

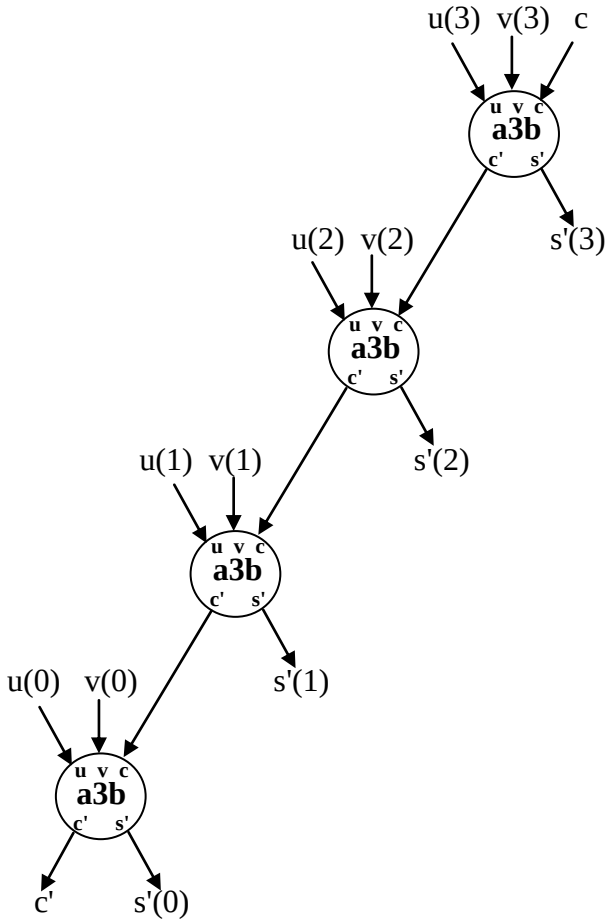
Let us now use symbolic execution on the set { c', s'=add(u, v, c, 3); } so as to get a block diagram of a circuit for adding bits in the range 0 to 3, that is a 4 bit adder. The execution is shown as a sequence of equivalent sets of statements as follows:

{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.

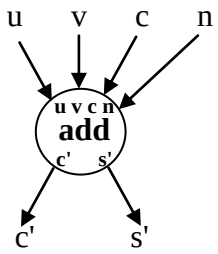
≡{c',s'(0)=a3b(u=u(0); v=v(0);
c,s'(1)=a3b(u=u(1); v=v(1);
c,s'(2)=a3b(u=u(2); v=v(2);
c,s'(3)=a3b(u=u(3); v=v(3); c=c))));}

The last set contains only hardware operations. It essentially describes the following 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 this diagram.



The textual presentation of the 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. So here is a diagrammatic description of the add function.

A function call such as c', s'=add(u, v, c, n); can be represented by the diagram:



This diagram is equivalent to one of the following diagrams depending whether or not n>0.

Diagram for n>0

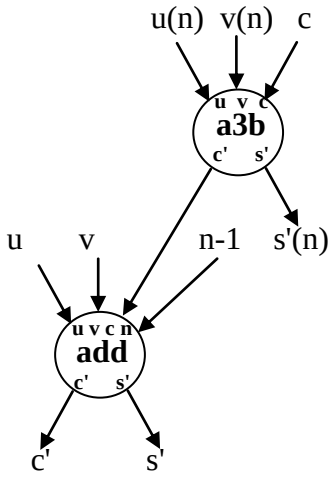
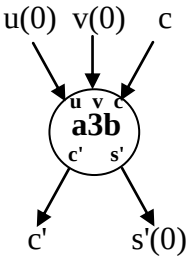
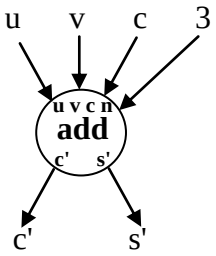


Diagram for n=0

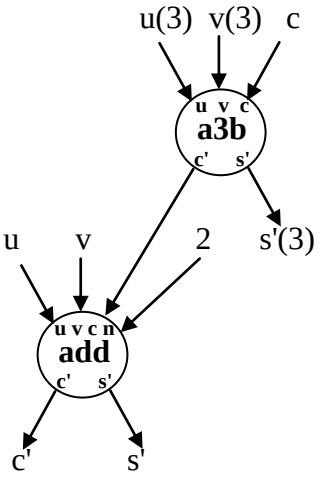


Let us now use symbolic execution in diagrammatic form so as to get a block diagram of a circuit for adding bits in the range 0 to 3, that is a 4 bit full adder.

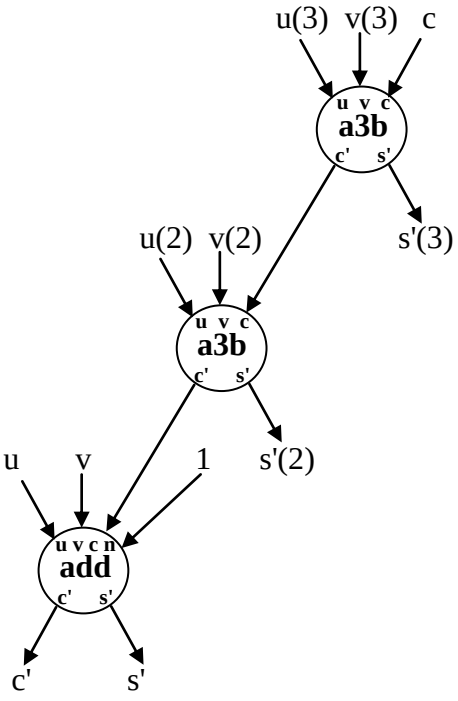
The statement c', s'=add(u, v, c, 3); can be represented by the diagram:



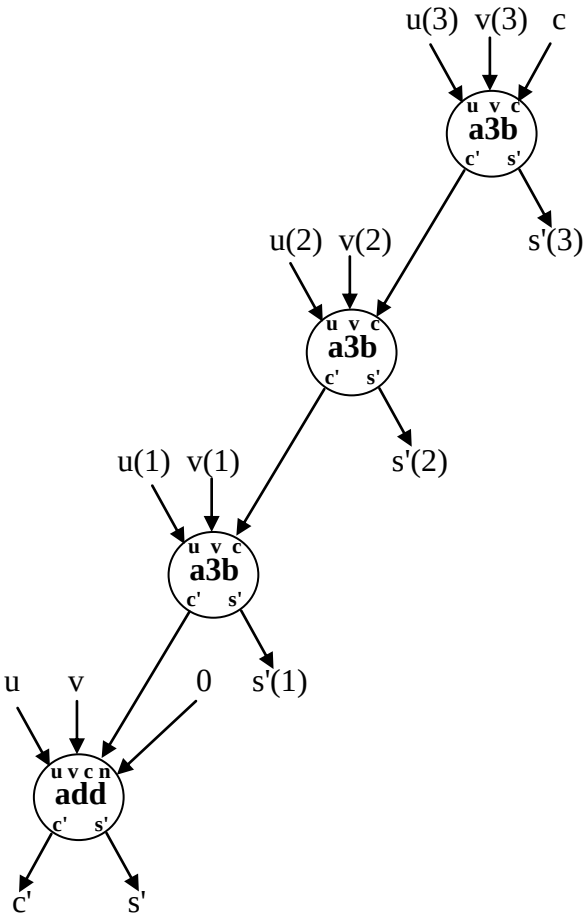
This is equivalent to the diagram:



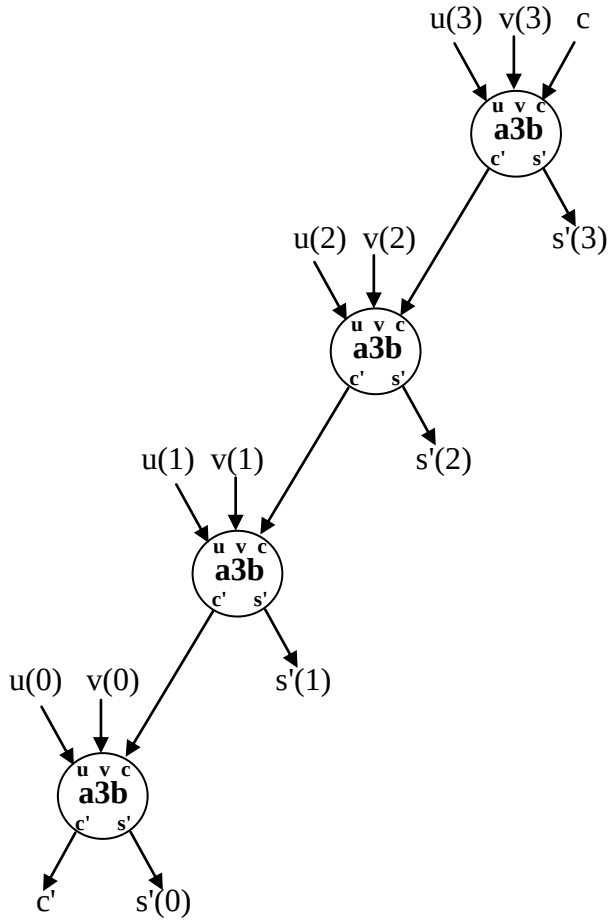
This is equivalent to the diagram:



This is equivalent to the diagram:



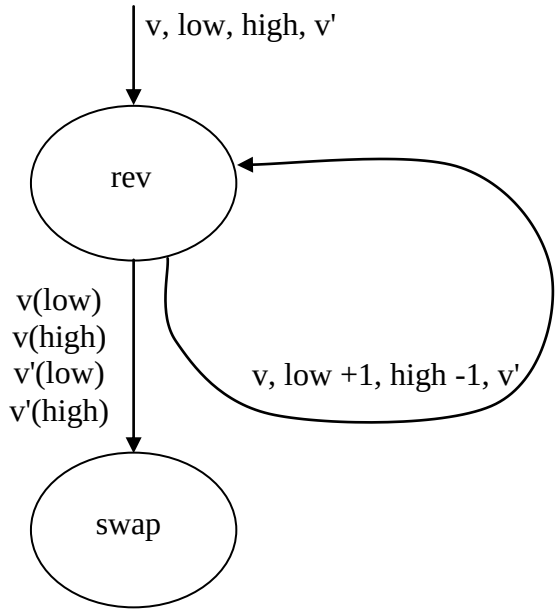
Finally, this is equivalent to the diagram:



This is exactly what we got from the symbolic execution presented in textual form.

DATA FLOW DIAGRAMS

Data flow diagrams, used in software design, can be produced from SFL programs by ignoring all operations except function calls of user defined function and showing this on the diagram. Here is a sort of data flow diagram for the functions "rev" and "swap", where for the sake of illustration we treat "swap" as a user defined operation.



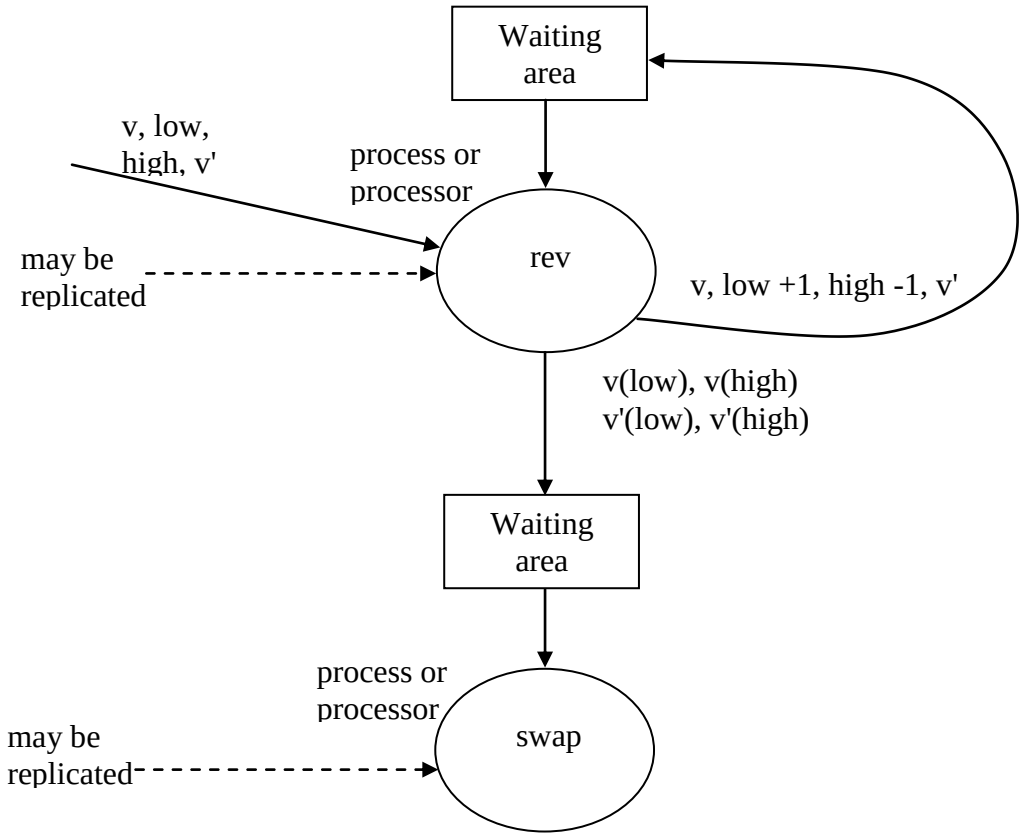
Similarly, a skeleton program containing calls only to user defined functions can be produced from a data flow diagram by using the data passed on arcs as parameters.

Here is a skeleton program for "rev" corresponding to the above data flow diagram. To complete the definition, basic operations need to be added and tests made to derive under what conditions the calls should be made.

```
/* SKELETON PROGRAM - PLEASE MODIFY AS NECESSARY */
/* SPECIFICATION:  PLEASE DEFINE */
function rev (IN v; int low, high;
              OUT vector v');
{v' = rev (v, low+1, high-1);
v'(low), v'(high) = swap ( v(low), v(high) );
}
```

ARCHITECTURE DIAGRAMS

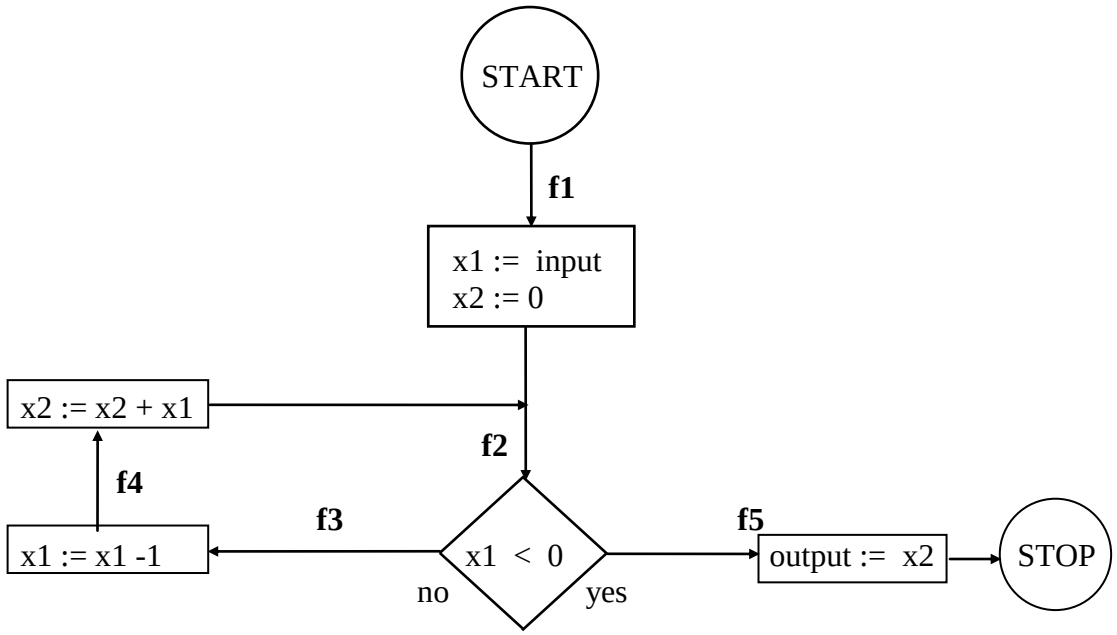
We can systematically produce an architecture diagram, similar to a data flow diagram, of a network of processes or processors for the example of rev by having a waiting area for handling calls to functions. Processors or processes attached to a waiting area may be replicated.



FLOWCHARTS AND SFL

Any simple flowchart program can be converted to a set of SFL function definitions as can be seen from the following example. In converting a flowchart to a set of SFL functions, we put different function names at the entry to each box in the flowchart,

and write functions reflecting the effect of running the program from that point in the flowchart. As can be seen from this example, the functions and flowcharts have a similar structure.




```
function f1 (IN int input; OUT int output')
{ output' = f2 (input , 0); }

function f2 (IN int x1, x2; OUT int output')
{ if (x1 < 0)
    output' =  f5 (x1 , x2);
    else output' =  f3 (x1 , x2); }

function f3 (IN int x1, x2; OUT int output')
{ output'  =  f4 (x1 - 1 , x2) }

function f4 (IN int x1, x2; OUT int output')
{ output'  = f2 (x1 , x2 + x1) }

function f5 (IN int x1, x2; OUT int output')
{ output'  = x2 }
```

Here is how execution of one iteration of the loop in the flowchart looks like when input=2.

place	input	x1	x2	output
f1	2			
f2		2	0	
f3		2	0	
f4		1	0	
f2		1	1	

Here is a similar computation sequence using the functions in SFL, up to the first recursive call of f2. Note the close correspondence with the foregoing.

```
f1(2)
=f2 (2, 0)
=f3 (2, 0)
=f4 (1, 0)
=f2 (1, 1)
```

CONVENIENT EXTENSIONS

The following, though not essential, are convenient extensions. We require that all such extensions can be converted to "core" SFL. This ensures that all the flexibility is retained.

Blocks and local variables

It is often useful to use a local variable to prevent re-computation of values. Suppose the value x+y-z is used several times in the definition of the function below.

```
function f(IN int x,y,z; OUT int r');
/* SPECIFICATION … */
{
/* statements of f */
… x+y-z…
…
… x+y-z…… x+y-z…
…
}
```

Then to avoid re-computing this value we can use a local variable in a block and write f as follows:

```
function new_f(IN int x,y,z; OUT int r');
:( int xyz=x+y-z )    /* Local variable(s) with their initial value. */
{
/* statements of f with xyz in place of x+y-z */
… xyz…
…
… xyz…… xyz…
…
}
```

Note even though this is not written explicitly, the local variable of a block is of IN type only and is given a value with its declaration. This is to allow conversion to "core" SFL. (This requirement is slightly relaxed in the section dealing with the user interface.)

Here is how this will look in "core" SFL without a block and local variable but with an auxiliary function having an extra IN parameter.

```
function new_f(IN int x,y,z; OUT int r');
{
r'=block(x,y,z,x+y-z);
}
function block(IN int x,y,z,xyz; OUT int r');
{ /* statements of f with xyz in place of x+y-z */
… xyz…
```

```
…
… xyz…… xyz…
…
}
```

This view treats a block as a function which has only one external call. This means of course that everything we have said so far about the flexibility of SFL will remain true if we add blocks and local variables, since this notation can be viewed as a convenient abbreviation.

Labeled blocks (Loops)

We allow blocks to be labeled (really a convenient abbreviation for a function definition). We can then execute blocks by using their names and if necessary provide values for local variables (really a function call). This notation allows us to write in a style similar to loops ("for", "while"). We require the same scope rules for labels as for functions, variables and other names. Here is how we can square every element in a vector using a labeled block (similar to a while loop).

```
function squares(IN vector v; OUT vector v');
/*
SPECIFICATION: v, v' are vectors having the same length. Each element of v' is the
square of the corresponding element of v.
*/
:loop (int i=0)
        {
          if (i<v.length)
             { loop(i=i+1); /* for next "iteration" */
               v'(i)=v(i)**2; };
        }; /* squares */
```

Note that "loop" is the name of the labelled block and "i" is its local variables. As with the block, the labelled block loop can be translated to "core" SFL by using an auxiliary function as follows.

```
function new_squares (IN vector v; OUT vector v');
{ loop(i=0); }; /* new_squares */

function loop(IN vector v; int i; OUT vector v');
{ if (i<v.length)
     { loop(i=i+1); v'(i)=v(i)**2; };
}; /* loop */
```

Again, this view treats a labeled block as a function which may have one or more external and internal calls. Again this addition will retain the flexible execution of SFL. In particular, parallel execution of several iterations of the loop is possible by using different local variables for each iteration.

(Though it is possible to rewrite blocks and labeled blocks as functions, it is sometimes more efficient not to do this, as blocks and labeled blocks have the potential for more efficient execution.)

Other features

We can add other features to SFL such as "for loops", "while loops", "case" statements etc. We must be careful however that these extra constructs are equivalent to compound constructs in "core" SFL so as to retain flexibility. For example, here is how a "for loop" could be written as a labeled block which of course can be converted into "core" SFL.

```
for (int i=0; i<n; i++)
{ statement list };
```

This can be rewritten as:

```
:for (int i=0)
{
 if (i<n)
   { for(i=i+1); statement list }
}
```

THE USER INTERFACE

The flexibility of SFL in allowing different orders of execution (and different methods of implementation) follows from the following properties of the language.

1) All variables are local variables (no permanent variables).
2) Variables may be assigned to once only (no update of variables).
3) Variables are either IN or OUT (no IN/OUT).

The traditional user interface does not comply with the above properties and so is highly sequential in use.

So using it with SFL programs would restrict the flexibility of execution. To deal with this problem, it is desirable to use SFL, with certain primitive SFL functions, as the command language (or shell) for handling the user interaction.

To work within this framework we need to view the user/computer system interaction as part of an ongoing computation jointly executed by the user and computer system, typically over many sessions. There will be no permanent variables, but the computer system will retain local variables which are still accessible or active. Permanent system objects such as a printer, keyboard, files will be represented by location attributes of data types and to access these devices, variables having these attributes need to be declared. In this way we can retain the three properties listed above, thus enabling flexible execution (sequential or parallel). The logout command suspends the user interaction and preserves values of active and accessible variables. (This approach allows, but does not obligate, processing to continue in the background). When the user logs on again, he returns to where he left off with the same variables.

As it is not possible to update variables, new variables with the changed values will need to be created. To handle the automatic creation and deletion of user accessible variables there is no need to add new features to the language, as all variables are local variables. So blocks and function definitions are adequate for this purpose. Here is an example of how files keyboard and printer can be accessed in an SFL like user interface.

```
{ IN float pi = 3.14159;
      int (keyboard) r;
OUT int (printer) d';
      float (printer) c';
      float (file f) a';

d' = 2*r;
```

```
c' = 2*r*pi;
a' = pi*r*r;
}
```

The printouts are independent of each other and would be on different printers (virtual printers or spool files).

Here is another way of handling this with one string variable located on a printer.

```
{ IN float pi = 3.14159;
        int (keyboard) r;
  OUT string (printer) s';
          float (file f) a';
     {  int  d = 2*r;
        float  c = 2*r*pi;
        s' = sprintf ("%d \n %d", d, c);
     }
  a' = pi*r*r;
}
```

Note: IN and OUT variables with location attributes may only be used in outermost blocks of the SFL shell.

See also our works [14, 15] on implicit communication and separating algorithm from communication on which the above approach is based.

THE PROGRAMMING ENVIRONMENT AND DEBUGGING

Here are some preliminary thoughts on these matters.

1) Sequential execution using a stack, with delayed calls, would provide a good environment for debugging. The programmer is encouraged to look at the overall execution of a function invocation, before jumping to (other) functions it may call, in a way similar to the computational induction proof method.

2) In certain situations one may run SFL programs without a run time check on once only assignment. Small programs, which have been fully debugged, proved correct and require high speed execution, can be run in this way. At present, large programs and systems can not in practice be proved correct and invariably contain bugs. So they should not be run without this run time check, no matter what the speed benifits.

3) The assignment like style of parameter passing and initializing local variables of blocks can help identify situations where a regular assignment can perhaps be used to save on storage. Consider the following call, from the body of function "mergesort" in APPENDIX 1.

```
merge( v=mergesort(v, low, split); v=mergesort(v, split+1, high);
          low1=low; high1=split; low2=split+1; high2=high; index=low )}
```

Unless v is being referenced or read outside the functions being defined in APPENDIX 1, the same storage may be used for the new variable v on the left hand sides of "assignment like statements" as for the current variable v on the right hand sides. So sometimes regular assignments can be used since the elements of current v are not referenced or read once the corresponding elements of the new v are assigned or written. During debugging, an element of the current v can be made inaccessible, once the corresponding element of the new v is assigned to help check if the same storage can indeed be used for new and the current variables.

TEACHING THESE CONCEPTS

Here is a tentative 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.

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.

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. Skeleton SFL programs can be produced from Data flow diagrams.
- 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 close 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-", "pH" and "SISAL" languages [7, 8, 16] handle it. 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.

Implementation issues

- More memory may be needed and there may be a need to handle the state of a variable (unassigned, assigned, in computation, value in error).
- There is a need for a mechanism to allow small changes to data, e.g. declarations such as: IN vector v; OUT vector v' = v exceptions 3;. This means that the (default) value of v' is like v but there are up to three exceptions which may be assigned later. This allows us to allocate separate storage for the exceptions. When v is no longer accessable, the memory management system can carry out assignments and use the storage of v for v' with no exceptions. We think this approach preserves flexibility of execution but more work needs to be done to confirm this. This feature allows us to have

several "versions" of a vector with a few differences between them and to store the original once and exceptions for the other "versions". (Sometimes it may be preferable to make the changes to v' and use an exception list for v, perhaps when the functions are tail recursive with single tails.)

- Similar to the previous point is to allow user definition of initial (default) values for OUT variables with their declaration, for example: OUT boolean flag'=false; meaning that initially flag'=false but may be assigned later. Multiple assignment is an error, but initialization followed by one assignment is not. We can in fact add language specified initial (default) values for all variables which would ensure that all unassigned variables are well defined. Flexibility of execution is preserved with this approach, but care must be taken that only the final value of an output variable is read or used in subsequent computations. See also APPENDIX 4 - Composite assignments. (A "lazy" version of this approach would be to evaluate and assign the default value only if the variable is unassigned. We do not take this "lazy" approach in SFL because of compatibility reasons with composite assignments. It also does not fit in well with the requirement that all computations should be performed.)
- An alternative assignment definition which allows flexible execution is not to treat multiple assignments of the same value as an error. While this may be acceptable for basic data types, for compound data such as vectors etc. this is inefficient. We think that more efficient programs will be developed if multiple assignments are treated as errors and this is the default approach taken in SFL. However, there are situations where the alternative is useful, and so we provide the alternative form of assignment as well and denote it by "e=". We also require that only one kind of assignment can be used with each variable. See also APPENDIX 4 - Composite assignments.

Other issues

With procedural programming, error handling is relatively easy, as only one error needs to be handled at a time. Flexible execution makes this a much harder problem to handle in SFL. Here are some preliminary thoughts on providing error handling mechanisms.

1) The result of evaluating a basic function returns a value and error flags. The result of evaluating an expression yields a value and the "or" of all the error flags of the functions activated. A value can be relied on only if all associated error flags are "off".
2) It is not clear if it is possible to incorporate the "throw, try, catch" error handling features of C/JAVA, since several errors may need to be caught by one "catch" because of parallel execution.
3) Default values are a possible means of error recovery.

Deadlock free? I believe yes, but don't have a formal proof. Certainly can't have in a single function a circular wait situation such as X = Y = Z = X as variables are either IN or OUT and here X is used as both IN and OUT.

Branch prediction for efficient use of an instruction pipeline is simpler in SFL in view of the fact that a variable is of type IN or OUT only (and is assigned to once only).

```
e.g.  if (f(x) > g(y))
        { …. };
      if (h(x) == k(y))
        { …. };
```

So since the variables x, y can not change in the above program fragment, branch predictions will be made correctly.

It would be interesting to determine if Petri nets, Grafcet notation and Synchronous systems can be converted to SFL and vice versa. Real time information can be made available to programs as the SFL shell allows declaring a variable as residing on a virtual device such as timers ports etc. Real time control is harder as the execution order is flexible. Perhaps real time control should not be provided through SFL itself, but through a companion configuration language or execution control language. (Some of the ideas in [14, 15] on implicit communication and separating algorithm from communication may be relevant here.)

Parallel execution without locking is possible if parameters are evaluated in parallel before a function call is executed (parallel call by value).

If we wish to evaluate the fuction call and its parameters in parallel then at times it will be necessary to wait for parameters to receive their values and a locking (or waiting) mechanism can be used. Variables would typically be locked against reading until they receive a value. Once they receive a value, they can be unlocked for reading and locked against writing. This gives early availability of variable values, and still allows detection of multiple assignment errors. In this way, a function composition such as f(g(x)) will execute with f and g being executed in parallel, and f waiting as needed for the values produced by g. (This implementation of function composition and parameter passing can provide something similar to the UNIX pipe and tee mechanisms.)

Besides once only assignment, certain composite assignments also allow flexible execution. For example if an OUT variable is given an initial value and all composite assignments perform a bitwise "or" to this variable, flexible execution is possible providing that care is taken so that a variable is only read after all composite assignments to it have completed. (This can similarly be done by using bitwise "and".) Indeed, any commutative and associative operation preferably with a unit element can be used, where the initial value of a variable is the unit element or some other standard or user provided value. See also APPENDIX 4 - Composite Assignments, where the associative/commutative condition is relaxed.

Perhaps surprisingly, we can have parameters of type INOUT *but not* with its usual meaning. We treat an INOUT parameter as an abbreviation for two parameters, one IN and the other OUT.

So for example: INOUT vector v*;*
really means: IN vector v; OUT vector v';
This approach retains the flexible execution of SFL, but we do not pursue this further here.

Variables may receive a value or written only once in SFL. Would there be any value to an IN1 variable meaning its value could only be referenced or read only once? In procedural programming languages, reading from a sequential file has this characteristic, the benefit being reduced storage needs for the program. In SFL, such a feature can also reduce memory requirements since in an assignment y'=x; if x is an IN1 variable, then its storage may be freed after the assignment, or in certain situations, the same storage can be used for y' and x there being no need for an assignment at all. At this stage we do not pursue this further here. See also [15] where we first raised the possibility of a variable of type IN1.

It is difficult to cast SFL in an object oriented style in view of the fact that variables are either "IN" or "OUT" and assignment is once only. Also all variables in SFL are local variables and this is in opposition to the use of global variables in objects and

classes. Perhaps it is possible to have objects whose variables are of type OUT and whose methods (functions) have parameters of type IN only but use the OUT variables of the object. Further work is needed here and for now a preliminary step in providing objects is described in APPENDIX 2 - Explicit Bindings.

Future projects

1) Software implementation of SFL (interpreter, compiler, debugger).
2) Man-machine interface and execution control language for SFL (text based, windows based).
3) OS support for SFL (virtual devices, virtual memory).
4) Hardware support SFL (instruction decode in advance, jump prediction, instruction pipeline, memory support).
5) Software for producing various kinds of block diagrams for SFL programs.
6) Software for producing skeleton program for DFD diagrams.
7) Program development support for SFL (easy access to specifications of functions called and support for program verification).
8) SFL for FPGA chips and PLC's.
9) Special programmable chips for SFL.

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] "The Esterel Synchronous Programming Language: Design, Semantics, Implementation", Gérard Berry and Georges Gonthier. Science of Computer Programming vol. 19, no. 2, pp 87-152, 1992.
[14] "Implicit Communication in Programming Languages by Declaration, Reference, and Assignment", with H.G. Mendelbaum, presented at Israel IEEE Computer Society - .4991 rebmeceD ts1 ot rebmevoN ht82 ,ytisrevinU nalI raB ,keeW gnitupmoC אי"ל
[15] "Towards Implicit Communication and Program Distribution", with H.G. Mendelbaum, CDROM Conference Proceedings, 4th World Multi-Conference on: Circuits, Systems, Communications and Computers (CSCC 2000), Vouliagmeni, Athens, Greece, July 2000. Also appears in "Advances in Physics, Electronics and Signal Processing Applications", pp. 402-409, edited by N. E. Mastorakis, World Scientific and Engineering Society Press, 2000.
[16] "SISAL 1.2: A Brief Introduction and Tutorial", David C. Cann, Research Report, Lawrence Livermore National Laboratory, May 1992.

APPENDIX 1 - "merge sort" in SFL.

```
function mergesort (IN vector v; int 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 sorted. Other elements of v' are not given values by this function.
If low>high, then the function does nothing to v'.
*/
{
if (low<high)
  :( int split=(low+high)/2) )
  {
   merge( v=mergesort(v, low, split); v=mergesort(v, split+1, high);
            low1=low; high1=split; low2=split+1; high2=high; index=low )
  }
else if (low==high)    /* Depending on the hardware, it may
                          be preferable NOT to use an else here. */
      {v'(low) = v (low);};
} /* end mergesort */

function merge (IN vector v; int low1, high1 low2, high2, index;  OUT vector v');
/*
SPECIFICATION:
IN - "v" is a vector and "low1", "high1", "low2", "high2" are positions within the vector v where we assume low1≤high1 and low2≤high2. In the ranges "low1" to "high1" and "low2" to "high2" the vector v is assumed to be sorted.
OUT - From the position "index", v' is like v in the ranges "low1" to "high1" and "low2" to "high2" but sorted.  Other elements of v' are not given values by this function.
*/
{
if (v(low1)<v(low2)
  { v'(index)=v(low1);
    if (low1<high1)
```

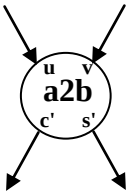
```
        {merge(low1=low1+1; index=index+1);}
      else {copy(low=low2, high=high2, index=index+1);}
    }
else { v'(index)=v(low2);
       if (low2<high2)
         {merge(low2=low2+1; index=index+1);}
       else {copy(low=low1, high=high1, index=index+1);}
     }
} /* end merge */

function copy (IN vector v; int low, high, index;  OUT vector v');
/*
SPECIFICATION:
IN - "v" is a vector and "low", "high" are positions within the vector v where we
assume low≤high.
OUT - From the position "index", v' is a copy of v in the range "low" to "high".  Other
elements of v' are not given values by this function.
*/
{ v'(index)=v(low);
  if (low<high)
     {copy(low=low+1; index=index+1);};
} /* copy */
```

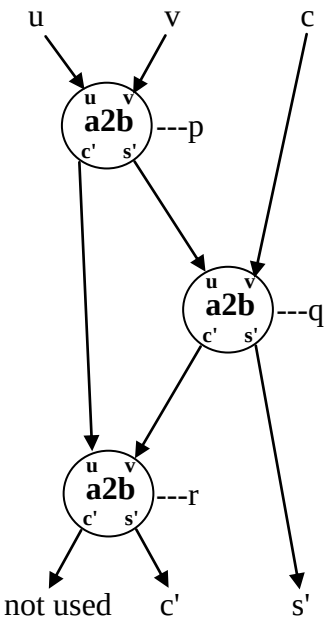
APPENDIX 2 - Explicit Bindings

Explicit bindings are a direct way of showing how values pass between various functions. This is explained with an example, where the arrows show how values are passed; i.e. how arguments are bound to the appropriate formal parameters.

Let us assume that there is a hardware operation "a2b" for adding two bits giving their carry and sum respectively (i.e. a half adder). This operation can be represented by the following diagram.



Let us now construct a full adder "a3b" for adding three bits giving their carry and sum respectively. Here is a block diagram for "a3b" where for the sake of illustration, we only use the operation "a2b" in the diagram.



Note that because of the particular situation above, the last operation "a2b ---r" may be replaced by an "or", which is the usual way of making the construction.

The above block diagram can be expressed in SFL using the features we have so far described as follows.

```
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.
*/
:( bit c1,s1=a2b(u,v) )
{
  :( bit c2,s2=a2b(s1,c) )
  {
    _,c'=a2b(c1,c2);    /* "_" on the left denotes an anonymous variable, */
                        /* i.e. a result returned from a2b is not used.    */
    s'=s2
  }
}; /*a3b*/
```

This is not as direct as the diagram and there is a need for additional variables. To overcome this limitation of the textual form we use the brackets "[ ]" to make bindings explicitly.

```
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.
*/
{
 [
  a2b p, q, r;     /* Like declaring three "objects" or "components" of "type" a2b. */
  p.u=u; p.v=v;
  q.u=p.s'; q.v= c;
  r.u=p.c'; r.v=q.c';
  c'=r.s'; s'=q.s';
 ]
}; /*a3b*/
```

Notes

1) In the function definition above, p, q, r correspond to the labels p, q, r in the diagram. We call a variable such as p.u a labeled input. We call a variable such as r.c' a labeled output.

2) All labeled inputs are required to receive a value but not all labeled outputs need be used. In the diagram and in the SFL definitions of "a3b", the output r.c' from the third "a2b" is not used.

3) The use of the underscore "_" above is similar to its use in Prolog to indicate an anonymous variable. An anonymous variable may not be read, that is, it may not be used on the right hand side of an assignment statement. Each occurrence of "_" is considered to be a distinct anonymous variable.

4) Bindings in programs may not be nested. This is not restrictive as any number of operations may be bound together in a single binding.

5) Note that in assignments, labeled input variables occur on the left and labeled output variables occur on the right. Also, in this example we started with a directed acyclic graph, and expressed it in textual form using explicit bindings. It is also possible to start with the textual form of explicit bindings and produce a directed graph showing the bindings. We require the graph to be acyclic and consider it a syntax error if this is not the case. This prevents a loop occurring in the graph of the explicit binding, which is important for avoiding circular wait situations i.e. deadlock.

6) Explicit bindings can provide some of the mechanisms of objects, including a form of inheritance as well as omission of components. However, further work is needed to design an object mechanism compatible with the flexible execution possibilities of SFL.

APPENDIX 3 - Introducing local variables to enable parallelism

Consider the following fragment of a *sequential* program in the style of C/Java.

```
{int i=5;
 /* statement list 1 */
 i=i+7;
 /* statement list 2 */
 i++;
 /* statement list 3 */
}
```

In converting this to SFL we could write

```
:( int i=5 )
{
 /* statement list 1 suitably modifed to single assignment form */
        :( int i=i+7 )
        {
         /* statement list 2 suitably modifed to single assignment form */
                :( int i=i+1 )
                {
                 /* statement list 3 suitably modifed to single assignment form */
                }
        }
}
```

So there are three variables called "i", and this allows the execution of the three modified statement lists to be overlapped. A similar approach can be used with labeled blocks (loops) so as to allow several "iterations" to be overlapped.

In the same spirit, new variables can be introduced by substituting on function calls. Here three substitutions have been made on the internal function call to "rev", giving the following equivalent form using nested blocks. (See the section "HARDWARE BLOCK DIAGRAMS", for the original definition of "rev".)

```
function rev  (IN vector v; int low, high;
                OUT vector v');
{
if (high > low)
{ v'(low), v'(high) = swap ( v(low), v(high) );
        :( int low, high=low+1, high-1 )
        {
        if (high > low)
        { v'(low), v'(high) = swap ( v(low), v(high) );
                :( int low, high=low+1, high-1 )
                {
                if (high > low)
                { v'(low), v'(high) = swap ( v(low), v(high) );
                v' = rev (v, low +1, high -1); }
                else if (high==low)
                { v'(high) = v(high)};
                }
        else if (high==low)
        { v'(high) = v(high)};
        }
else if (high==low)
{ v'(high) = v(high)};
}
 /* end rev */
```

Note the use of new local variables "low", "high" in the inner blocks which enables overlapping the execution of these blocks.

APPENDIX 4 - Composite Assignments

Languages such as "C" or "Java" allow composite assignments of the form "x f= y", meaning x=(x f y), where f is a binary operator or function of two arguments. A typical example is x += y. It also supports abbreviated forms of composite assignments such as x++ meaning x += 1 or x=(x + 1). Can composite assignments (and their abbreviations) be incorporated into SFL while retaining flexible execution? For which kinds of functions and situations is this possible? Solutions to these questions are important as they will allow programmers to use the familiar algorithmic style, and they will enable flexible execution with reduced storage requirements. (The approach taken here differs from the approach of the previous appendix, in that we wish to allow flexible execution for certain composite assignments, *without* the need for multiple copies of a variable.)

For example, consider the function:

function g (IN ...; OUT some_data_type x'=e_0;) /* initial value for x' */
{ x' f= e_1;
...
x' f= e_n;
}

Here e_0, ..., e_n are expressions and f is as above. Here are some conditions, which enable considerable execution flexibility while ensuring a uniquely defined final result for x'.

1) It is required that ((a f b) f c) = ((a f c) f b). While the first parameter and the value of "f" must be of the same type, no restriction is placed on the second parameter of "f". Examples of functions satisfying this condition are "and", "or", exact arithmetic operations +, -, *, /, **, addition and subtraction with respect to a fixed modulus, etc. (Associative/commutative conditions or the existence of a unit element are not required.)

2) In this specific case, e_0, ..., e_n may be evaluated sequentially or in parallel, in any order. (In other situations it can be necessary to evaluate e_0 first.)

3) The initialization x'= value of e_0, must be carried out before the composite assignments.

4) The composite assignments x' f= value of e_1; ...; x' f= value of e_n; must be performed sequentially <u>but in any order</u>.

5) Care must be taken that only the final value of an output variable is read or used in subsequent computations.

For now we assume that only one kind of assignment can be used with each variable. Later we will relax this condition.

The final value of x', equals value of the expression (...((((e_0 f e_{i_1}) f e_{i_2}) f e_{i_3}) ... f e_{i_n}) in an execution according to the above. Here i_1, i_2, ... i_n are a permutation of 1, 2, ... n and are determined by the order in which the composite assignments are performed.

Condition (1) allows consecutive e's in the aforesaid expression, except for e_0, to be swapped, without altering the value of the expression. As it is possible to sort by swapping consecutive elements, we can rearrange e_{i_1}, e_{i_2}, ... e_{i_n} to e_1, e_2, ... e_n by sorting on the subscripts of the e's using "bubble sort" for example. So this means that the final value equals the value of (...((((e_0 f e_1) f e_2) f e_3) ... f e_n) and so is uniquely defined.

Furthermore, we shall show that condition (1) can not be relaxed any further. Suppose that there are only two composite assignments and the final result is uniquely defined. If the first is made before the second, then the final result will be ((e_0 f e_1) f e_2). But if the second is made before the first the final result will be

$((e_0\ f\ e_2)\ f\ e_1)$.

As the final result is uniquely defined, it follows that $((e_0\ f\ e_1)\ f\ e_2) = ((e_0\ f\ e_2)\ f\ e_1)$. Apart from the names used, this is condition (1).

(Incidentally, conditions similar to the above apply for detecting violations of once only assignments in SFL during a parallel execution. Expression evaluation may be in parallel but assignments to a given variable must be sequential so as to guarantee detection of such violations. In a sequential implementation, this difficulty does not arise.)

Note: Composite assignments where the function satisfies condition (1), bear similarity to certain features of synchronous systems [13].

Handling composite assignments with several functions

Consider a more general case:

function g (IN ...; OUT some_data_type x'=e_0;) /* initial value for x' */
{ x' f_1= e_1;
...
x' f_n= e_n;
}

where f_1,...f_n are functions, which may or may not be different. (They may even all be the same, which is the case discussed above.) Condition (1) needs to be modified as follows:

1) It is required that $((a\ f_i\ b)\ f_j\ c) = ((a\ f_j\ c)\ f_i\ b)$ for $i \neq j$.

A similar argument to the above shows that the final value of x' equals the value of $(...(((e_0\ f_1\ e_1)\ f_2\ e_2)\ f_3\ e_3)\ ...\ f_n\ e_n)$ and is uniquely defined, subject to the other conditions enabling flexible execution. (In the sorting on the subscripts described above, the only change is to swap consecutive function expression pairs in the sort process and not just consecutive expressions.)

Allowing any composite assignment by restricting flexibility

We can allow the use of any composite assignment by restricting the flexibility. For example, the expressions on the right hand sides of the above assignment statements may be executed in parallel but the composite assignments would be executed sequentially in the order they are written. This ensures well defined read values providing all composite assignments to a variable are executed before its value is read. This is an interesting possibility as it provides a simple interface between sequential and parallel execution. Further study of this topic is needed, in particular, the efficiency of an implementation.

APPENDIX 5 - Generalized call statement and scope rules

The generalized call statement uses the assignment style of parameter passing to provide additional possibilities. This statement has three components: an entry block followed by the function to be called followed by an exit block. We explain this further by example.

Suppose we wish to call a function f whose IN parameter is a vector v, and whose OUT parameter is a vector v'. Suppose that we wish to pass to f a vector of a hundred elements in which v(i) is i^2 and receive the result in a'. The usual way would be to declare a vector sq, put the squares in sq, and apply f to sq. If we were to use the assignment style of parameter passing we could write at length:

f(v'=a'; v(0)=0*0; v(1)=1*1; v(2)=2*2; ...; v(99)=99*99;)

Here is a briefer and clearer way of doing this using the generalized call statement.

```
(call
  :loop(int i=0)                                   /* Entry block */
  { if (i<100) {loop(i=i+1); v(i)=i*i} }
  f                                                /* Function to be called */
  {a'=v'}                                          /* Exit block */
)
```

Here is something similar with OUT parameters. Suppose we wish to apply f to a vector "u" and put the first fifty elements of f(u) in a' and the last fifty elements in b'. Here is how this can be done without declaring an auxiliary vector.

```
(call
  {v=u}                                            /* Entry block */
  f                                                /* Function to be called */
  :loop (int i=0)                                  /* Exit block */
  { if (i<50) { loop(i=i+1); a'(i)=v'(i); b'(i)=v'(i+50) } }
)
```

Scope rules for IN and OUT parameters handled this way

The entry block would be typically executed before the function call. The exit block would be typically executed after the function results are evaluated. Note that as this is a call, the IN formal parameters may only be assigned in the entry block and the OUT formal parameters may only be referenced referenced in the exit block. (The OUT formal parameters are not accesible in the entry block and the IN formal parameters are not accesible in the exit block.) This is the opposite to what is allowed in the body of the function.

This notation avoids the need to declare and initialize an additional vector. It is also suggestive, in that the entry and exit blocks may be executed either by the process which executes f or by the process which makes the call.

Generalized call statements may be nested.

ACKNOWLEDGEMENTS

Sincere thanks to H. Dayan, H.G. Mendelbaum, S. Mizrahi, M. Reif and Isaac Dayan for their help in preparing these notes. Sincere thanks to Leyora Hellmann for her help in typing these notes.

R.B. Yehezkael (formerly Haskell).
Revised February 2013 - שבט תשע״ג.
(Minor corrections December 2015 – טבת תשע״ו)
Jerusalem College of Technology - Machon Lev,
Hawaad Haleumi 21, Jerusalem 91160, ISRAEL.
Tel: 02-6751111.
e-mail: rafi@jct.ac.il.