February 2007 (vol. 8, no. 2), art. no. 0702-o2001
1541-4922 © 2007 IEEE
Published by the IEEE Computer Society

Education

Flexible Algorithms: Fragments from a Beginners' Course

R.B. Yehezkael

In a previous column (http://csdl2.computer.org/comp/mags/ds/2006/11/oy002.pdf), I gave an overview of a beginner's course on flexible algorithms that my colleagues and I developed at the Jerusalem College of Technology. Here I briefly describe this notation's main features and present fragments from this course.

Our algorithmic notation

The core algorithmic notation of flexible algorithms is based on functions with In parameters and Out parameters (but no In/Out parameters) and conditional statements. As in mathematics, variables and parameters receive a value only once. Blocks and loops (including nested forms) aren't part of the core language but are abbreviations for certain compound forms in the core language. Our notation has an iterative style and includes a once-only assignment statement.

Because our notation can't update variables and parameters, it enables parallel execution.

Fragments from the course

The course notes (www.rby.name) present examples of flexible algorithms and their execution by three methods that use the same sets and values form:

- parallel execution,
- sequential execution with immediate execution of function calls or activations, and
- sequential execution with delayed execution of function calls or activations.

To save space, I'll only discuss parallel execution here.

Here I present some fragments from the course material.

Reversing five values

The following is a flexible algorithm for reversing five values.

```
Function a′, b′, c′, d′, e′ •= rev5(a, b, c, d, e);

// Specification:
// In - a, b, c, d, e are any values.
// Out - The values a′, b′, c′, d′, e′ are like a, b, c, d, e but in reverse
order.
```

```
{  a′ •= e;
   b′, c′, d′ •= rev3(b, c, d);
   e′ •= a;
} // end rev5

function a′, b′, c′ •= rev3(a, b, c);

// Specification:
// In – values a, b, c
// Out – values a′, b′, c′ like a, b, c but in reverse order

{  a′ •= c;
   b′ •= rev1(b);
   c′ •= a;
} // end rev3

function a′ •= rev1(a);

// Specification:
// In – value a
// Out – value a′ like a but in reverse order—that is a′ is a

{  a′ •= a;
} // end rev1
```

Suppose we wish to execute the statement `a′b′c′d′e′ •= rev5 (1, 2, 3, 4, 5).` We begin by writing this as a set of one element, namely `{a′, b′, c′, d′, e′ •= rev5(1, 2, 3, 4, 5) }`. The steps of the parallel execution are given below.

```
set of statements                                          a′,b′,c′,d′,e′
{a′, b′, c′, d′, e′ •= rev5(1, 2, 3, 4, 5) }              _, _, _, _, _
{a′ •= 5 ; b′, c′, d′ •= rev3(2, 3, 4) ; e′ •= 1 }        _, _, _, _, _
{b′ •= 4, c′ •= rev1(3) ; d′ •= 2 }                       5, _, _, _, 1
{c′ •= 3 }                                                 5, 4, _, 2, 1
{ }                                                        5, 4, 3, 2, 1
```

For educational reasons, we presented a specific solution for five values before giving a function for reversing part of a vector. We also presented three execution methods for reversing a five-element vector using this general solution. See the course notes for details.

Avoiding errors

Two kinds of errors can occur. Syntax errors occur in the flexible algorithm's form or syntax, making executing the algorithm pointless. (In other words, you don't need to execute the algorithm to see that an error exists.) Execution (or *runtime*) errors occur when a flexible algorithm is executed.

The following is an example of a syntax error:

```
function x′, y′ •= bug1(x, y);
{
   x′ •= 3;   // No error here.
   x •= y + 1;       // x may not be assigned here
                     // as it is an Input variable.
   y′ •= x′ + 1;     // May not use the value of x′ here
                     // as it is an Output variable.
}
```

Other syntax errors include brackets that don't balance, punctuation errors, and so forth.

The following is an example of an execution error:

```
function x′ •= bug2(x);    // No error in the form or syntax of the flexible
algorithm.
{
  if (x ≤ 3)     x′ •= x + 1;
  if (x ≥ 3)     x′ •= x + 2;
}
```

However, if the value of *x* is 3, an error occurs during execution, because two assignments are made to *x*′ (a conflict).

Avoiding these errors has the benefit of encouraging students to write in a style that enables but doesn't force parallel execution.

Summing eight numbers

The statement `s′ •= a0 + a1 + … a7` is equivalent to `s′ •= ((…(a0 + a1) + a2) + a3) + … a7)` when fully bracketed. Bracketing the statement in this way forces sequential evaluation. Rebracketing the expression in the following statement enables parallel evaluation:

```
s′ •= (((a0 + a1) + (a2 + a3)) + ((a4 + a5) + (a6 + a7)))               (1)
```

Intuitively, the calculation of `s′` in this way is performed as in figure 1.

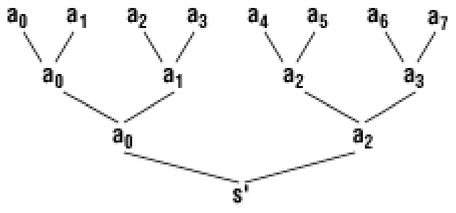

Figure 1. The calculation of s′ using statement (1).

In figure 1, it's useful to view each occurrence of a_i as a separate variable. This view simplifies writing the calculation as a flexible algorithm in which subexpressions can be computed in parallel. So, for example, there are three variables named a_0 and so forth.

Here are three functions for carrying out the computation in this way:

```
function s′ •= add8 (a0, a1, a2, a3, a4, a5, a6, a7)

// Specification: s′ = a0 + a1 + … a7

{add4 (a0 •= a0 + a1; a1 •= a2 + a3; a2 •= a4 + a5; a3 •= a6 + a7);}

// end add8


function s′ •= add4(a0, a1, a2, a3);

// Specification: s′ = a0 + a1 + a2 + a3

{add2(a0 •= a0 + a1); a1 •= a2 + a3);}

// end add4


function s′ •= add2(a0, a1);

// Specification: s′ = a0 + a1

{s′ •= a0 + a1};

// end add2
```

Here's the parallel execution of `s′ •= add8(1, 2, 3, 4, -1, -2, -3, -4).`

```
set of statements                                        s′

{s′ •= add8(1, 2, 3, 4, -1, -2, -3, -4)}                 _

{s′ •= add4(3, 7, -3, -7)}                               _

{s′ •= add2(10, -10)}                                    _

{ }                                                      0
```

Summing the elements of a vector

Here's how we can generalize the previous example to handle a vector of *n* elements, where *n* is a power of 2. (A simple exercise that the students later perform is generalizing this solution so that it works for vectors of any size, not just a power of 2.)

```
function s′ •= addn(v,n)
// Specification:v is a vector and n is its size which must be a power of 2.
// The function computes s′ = v0 + v1 … + vn−1
{
if      (n = 1)
        {s′ •= v0}
else addn (
                           n •= n/2;
                           v0 •= v0 + v1;
                           v1 •= v2 + v3
                           .
                           .
                           .
                           vn/2−1 •= vn−2 + vn−1
                 );
} // end addn
```

(You can replace the lines from `v0 •= v0 + v1; to vn/2−1 •= vn−2 + vn−1` with a `forall` loop; see the course notes for details.)

Here's the parallel execution of `s′ •= addn((1, 2, 3, 4, −1, −2, −3, −4), 8).`

```
set of statements                                        s′
{s′ •= addn((1, 2, 3, 4, −1, −2, −3, −4), 8)}            _
{s′ •= addn((3, 7, −3, −7), 4)}                          _
{s′ •= addn((10, −10), 2)}                               _
{s′ •= addn((0), 1)}                                     _
{ }                                                      0
```

Another simple exercise is to write a flexible algorithm to find a vector's minimum value using the structure of the function `addn`. The students need only to replace the operator `+` with a function `min` for determining the minimum of two values and then write the definition of the function `min`.

Merge sort

We use the structure of the function `addn` to develop the bottom-up merge-sort algorithm. This gives students a gentle introduction to a subtle sorting technique that enables parallel sorting of a vector.

This technique sorts a vector by repeated merging. For simplicity, we assume that the vector's length is a power of 2. For example, suppose we wish to sort a vector of eight elements. We view this as eight single elements, and each single element is sorted. Here, for example, is a vector of eight elements:
`(8, 1, 7, 2, 6, 3, 5, 4)`
We merge pairs of single elements to obtain
`(1,8, 2,7, 3,6, 4,5)`
We now have four sorted pairs, and we merge two and two pairs to obtain
`(1,2,7,8, 3,4,5,6)`
This gives us two sorted runs of four elements, which we merge to obtain a sorted vector:
`(1, 2, 3, 4, 5, 6, 7, 8)`

At each stage, the sorted run in the vector doubles in size from the previous stage. The number of sorted runs in the vector is halved with respect to the previous stage (see table 1).

Table 1. Number and size of sorted runs.

No. of sorted runs	Size of sorted run
8	1
4	2
2	4
1	8

The number of sorted runs is like the number of values *n* to be added in the `addn` function. They would both progress 8, 4, 2, 1 when there are eight values.

Suppose we have a function `m2` with specification as follows.

```
function v'•=m2 (v, size, place);

// Specification:

// v is a vector having two ranges which are sorted

// in nondecreasing order.

// These ranges are of length "size" and at position

// "place" onwards in v.
// These two ranges are merged into a single range
// of length “2´size” and are put at position
// “place” onwards in v’.
```

Let’s now use the function `m2` to define the function `mergesort`.

```
function v’•=mergesort(v)
// Specification:
// v, v’ are vectors having the same length which
// must be a power of 2.
// v’ will be like v but sorted in nondecreasing order.

{loop(size•=1);}

function v’•=loop(v, size)
// Specification:
// v, v’ are vectors having the same length which
// must be a power of 2.
// The vector v consists of consecutive sorted ranges
// of length “size” which must be a power of 2.
// v’ will be like v but but sorted.
{
 if (size=length of v)
    {v’•=v}
   else
      loop (
                size•=size´2,
                v•=m2(v,size,0);
                v•=m2(v,size,2´size);
                v•=m2(v,size,4´size);
                v•=m2(v,size,6´size);
                .
                .
                .
                v•=m2(v,size,length of v – (2´size)
                );
 }
```

(You can replace the lines from `v•=m2(v,size,0);` to `v•=m2(v,size,length of v – (2´size)` with a `forall` loop; see the course notes for details.)

Converting flexible algorithms to hardware block diagrams

The course notes contain both a textual (hard to understand) and diagrammatic (easy to understand) development of a hardware block diagram for a serial adder, including its function definition in both textual and diagrammatic forms. (They also include an example of parallel execution in textual form.) Here, we only use the diagrammatic approach.

We assume that there's a function `a3b` for adding three digits that produces two results—a sum digit and a carry digit. For example, executing `c', s' •= a3b(9, 3, 1)` would make `s' •= 3` and `c' •= 1`. (Here `s'` is the sum and `c'` is the carry, each being a single digit.) Figure 2 shows a diagrammatic representation of `a3b`.

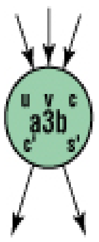

Figure 2. A diagrammatic representation of `a3b`.

The function `add` adds two numbers consisting of several digits and an existing single-digit carry. It gives two results, a sum consisting of several digits and a carry consisting of one digit. We assume that both numbers being added and the sum produced consist of n + 1 digits, where n denotes the position of the least significant digit. The most significant digit has position zero.

Figure 3a represents a function call `c', s' •= add(u, v, c, n)`. We can write equivalent diagrams for this function (see figures 3b and 3c).

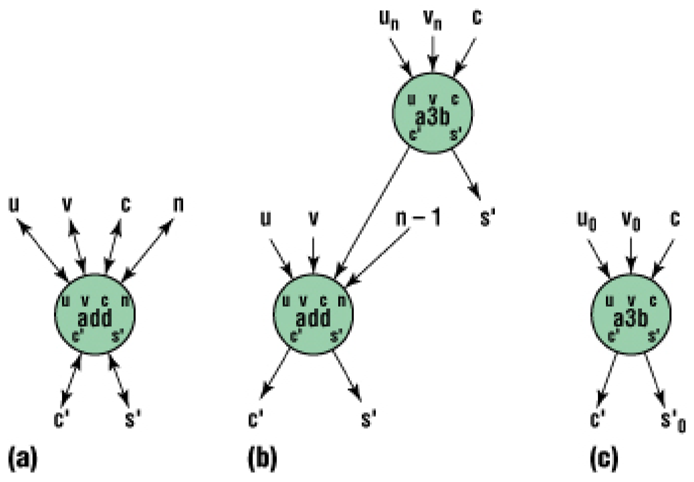

Figure 3. (a) A function call `c', s' •= add(u, v, c, n)` is equivalent to different diagrams (b–c), depending on whether n > 0 or not.

The call for a 4-digit adder is `c', s' •=add(u, v, c, 3)` (see figure 4a), which is equivalent to several other diagrams (see figures 4b–d) before resulting in the final diagram (see figure 4e).

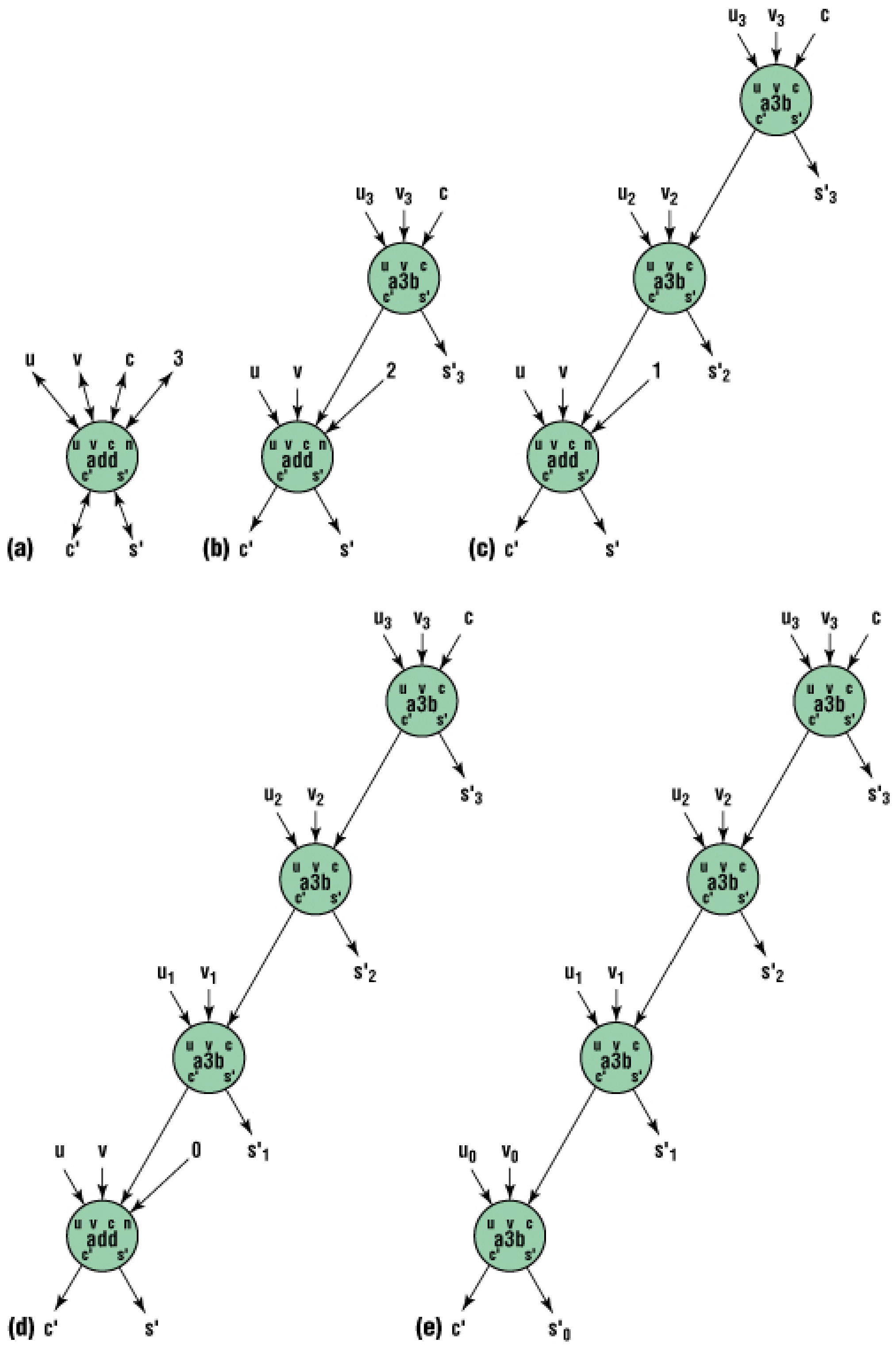

Figure 4. A 4-digit adder: (a) the call, (b–d) equivalent diagrams, and (e) the final diagram.

These fragments of the course material are all I can present in the space available, but I hope they give you a good idea of our approach. In the future, I hope to report on using this approach in secondary schools. I also hope to develop material for advanced students, such as the conversion of a flexible algorithm into a parallel algorithm where the parallelism is explicit.

Acknowledgments

This column is based on “Flexible Algorithms: Selections from a Course for Beginners,” a presentation I gave at the 4th IEEE International Conference on Information Technology: Research and Education.

R.B. Yehezkael (formerly Haskell) is an independent academic who retired from the Jerusalem College of Technology. Contact him at rafi@jct.ac.il; http://cc.jct.ac.il/~rafi.

Related Links

- DS Online's Education Archives
- "Report from the Ubicomp Education Workshop," IEEE Pervasive Computing
- "From Research to Classroom: A Course in Pervasive Computing," *IEEE Pervasive Computing*
- "An Undergraduate Success Story: A Computer Science and Electrical Engineering Integrative Experience," *IEEE Pervasive Computing*

Cite this article:

R.B. Yehezkael, "Flexible Algorithms: Fragments from a Beginners' Course," *IEEE Distributed Systems Online*, vol. 8, no. 2, 2007, art. no. 0702-o2001.