

# Alk Primer

Dorel Lucanu

March 17, 2016

# Chapter 1

## Introduction

### 1.1 Motivation

Alk is an algorithmic language intended to be used for teaching data structures and algorithms using an abstraction notation (independent of programming language).

The goal is to have a language that:

- is simple to be easily understood;
- is expressive enough to describe a large class of algorithms from various problem domains;
- is abstract: the algorithm description must make abstraction of implementation details, e.g., the low-level representation of data;
- is a good mean for learning how to algorithmically think;
- supply a rigorous computation model suitable to analyse algorithms;
- is executable: the algorithm can be executed, even if they are partially designed;
- is accompanied by a set of tools helping to analyse the algorithm correctness and the efficiency;
- input and output are given as abstract data types, ignoring implementation details.

As a starting example we consider the Alk description of the Euclid algorithm:

```
gcd(a, b)
{
  while (a != b) {
    if (a > b) a = a - b;
    if (b > a) b = b - a;
  }
  return a;
}
```

The algorithm is described using a C++-like notation. The name of the algorithm is `gcd` and its input parameters are `a` and `b`. There is no need to declare the type of parameters and/or the type of the return value. In order to execute the `gcd` algorithm, just add a single line algorithm

```
x = gcd(12, 8);
```

and execute it (`gcd.alk` is the file include the above code):

```
> alk gcd.alk
x |-> 4
```

The output is the final configuration of the execution, that includes the values of the global variables (here  $x$ ). An alternative is to write a general call of the algorithm

```
x = gcd(u, v);
```

and execute it by mentioning the initial values of the global variables  $u$  and  $v$ :

```
> alk gcd.alk -init="u |-> 42 v |-> 56"
```

The state from the final configuration:

```
u |-> 42
v |-> 56
x |-> 14
```

A more complex algorithm is the DFS traversal of a digraph represented with external adjacent lists:

```
dfsRec(i, out S) {
  if (S[i] == 0) {
    // visit i
    S[i] = 1;
    p = (D.a)[i];
    while (p.size() > 0) {
      j = p.topFront();
      p.popFront();
      dfsRec(j, S);
    }
  }
}
```

```
// the calling algorithm
```

```
dfs(D, i0) {
  i = i0;
  while (i < D.n) {
    S[i] = 0;
    i = i + 1;
  }
  dfsRec(i0, S);
  return S;
}
reached = dfs(D, i0);
```

The execution of the above algorithm on the digraph:

$$\begin{aligned} D.n &= 3, \\ D.a[0] &= \langle 1, 2 \rangle \\ D.a[1] &= \langle 2, 0 \rangle \\ D.a[2] &= \langle 0 \rangle \end{aligned}$$

is obtained as follows:

```
> alk dfsrec.alk -cINIT="D |-> { n -> 3
a -> [ < 1, 2 >, < 2, 0 >, < 0 > ] }
i0 |-> 1"
```

The state from the final configuration:

```
D |-> { (n -> 3) (a -> ([ < 1, 2 >, < 2, 0 >, < 0 > ])) }
i0 |-> 1
reached |-> [ 1, 1, 1 ]
```

## 1.2 Executing Alk algorithms with K Tools

The Alk language was designed using K Framework (<http://www.kframework.org/>). So we can use K tools for executing algorithms. The K definition of Alk is compiled using the `kompile` tool:

```
> kompile alk.k
```

Then the compiled definition can be used to execute Alk algorithms:

```
> krun gcd.alk -cINIT=".Map"
<k>
    .K
</k>
<state>
    x |-> 4
</state>
```

The initial state is given as the value of the option `-cINIT`; `.Map` is the K notation for the empty state. A nonempty state is specified as expected:

```
> krun gcd.alk -cINIT="u |-> 42 v |-> 56"
<k>
    .K
</k>
<state>
    u |-> 42
    v |-> 56
    x |-> 14
</state>
```

Here is the execution of the DFS algorithm with the K Tool:

```
> krun dfsrec.alk -cINIT="D |-> { n -> 3
    a -> [ < 1, 2 >, < 2, 0 >, < 0 > ] }
    i0 |-> 1"
<k>
    .K
</k>
<state>
    D |-> { (n -> 3) (a -> ([ (< 1, 2 >), (< 2, 0 >), (< 0 >) ])) }
    i0 |-> 1
    reached |-> [ 1, 1, 1 ]
</state>
```

**Remark.** *This is a in progress document that is incrementally updated.*

## Chapter 2

# Language Description

### 2.1 Values

Alk includes two categories of values: 3ex

*Scalars (primitive values).* Here are included the booleans, integers, rationals (floats), and strings.

*Structured values.* Here are included the sequences (linear lists), arrays, structures.

Note that a data can be as complex as possible, i.e, we may have arrays of sequences, arrays of arrays, sequences of arrays of structures, structures of arrays and lists, and so on.

#### 2.1.1 Scalars

The scalars are written using a syntax similar to that from the most popular programming languages:

```
index = 234;
isEven = true;
radius = 21.468;
name = "john";
```

The execution of the above algorithm produces the following output:

```
index |-> 234
isEven |-> true
radius |-> 2.1468e+01
name |-> "john"
```

Note that for the floating point numbers the scientific notation is used.

#### 2.1.2 Arrays

An array value is written as a sequence surrounded by square brackets:  $[v_0, \dots, v_{n-1}]$ , where  $v_i$  is a value, for  $i = 0, \dots, n - 1$ . Here is a very simple algorithm handling arrays:

Algorithm	Output
<pre>a = [3, 5, 6, 4]; i = 1; x = a[i]; a[i+1] = x;</pre>	<pre>a  -&gt; [ 3, 5, 5, 4 ] i  -&gt; 1 x  -&gt; 5</pre>

The multi-dimensional arrays are represented a arrays of arrays:

```

a = [ [ 1, 2, 3 ], [ 4, 5, 6 ] ];
b = a[1];
c = a[1][2];
a[0] = b;
a[1][1] = 89;

w = [ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 5, 6 ], [ 7, 8 ] ] ];
x = w[1];
y = w[1][0];
z = w[1][0][1];
w[0][1][0] = 99;

```

The output:

```

a |-> [ ([ 4, 5, 6 ]), ([ 4, 89, 6 ]) ]
b |-> ([ 4, 5, 6 ])
c |-> 6
w |-> [ ([ ([ 1, 2 ]), ([ 99, 4 ]) ]), ([ ([ 5, 6 ]), ([ 7, 8 ]) ]) ]
x |-> ([ ([ 5, 6 ]), ([ 7, 8 ]) ])
y |-> ([ 5, 6 ])
z |-> 6

```

## 2.2 Sequences (linear lists)

A sequence value is written in a similar to an array, but using angle brackets:  $\langle v_0, \dots, v_{n-1} \rangle$ , where  $v_i$  is a value, for  $i = 0, \dots, n-1$ . The list of operations over sequences includes:

<code>empty()</code>	returns the empty list $\angle$
<code>L.topFront()</code>	returns $v_0$
<code>L.topBack()</code>	returns $v_{n-1}$
<code>L.lookup(i)</code>	returns $v_i$
<code>L.insert(i, x)</code>	returns $\langle \dots v_{i-1}, x, v_i, \dots \rangle$
<code>L.remove(i, x)</code>	returns $\langle \dots v_{i-1}, v_{i+1}, \dots \rangle$
<code>L.size()</code>	returns $n$
<code>L.popFront()</code>	returns $\langle v_1, \dots, v_{n-1} \rangle$
<code>L.popBack()</code>	returns $\langle v_0, \dots, v_{n-2} \rangle$
<code>L.pushFront(x)</code>	returns $\langle x, v_0, \dots, v_{n-1} \rangle$
<code>L.pushBack(x)</code>	intoarce $\langle v_0, \dots, v_{n-1}, x \rangle$
<code>L.update(i, x)</code>	returns $\langle \dots v_{i-1}, x, v_{i+1}, \dots \rangle$

Example:

Algorithm	Output
<pre> 11 = &lt; 8, 3, 9, 4, 5, 4 &gt;; i = 1; x = 11.at(i + 1); y = 11.topFront(); 11.insert(2, 22); 11.update(3, 33); 12 = 11; 12.removeAt(0); 12.removeAt(3); 12.removeAllEqTo(4); </pre>	<pre> i  -&gt; 1 12  -&gt; &lt; 3, 22, 33, 5 &gt; 11  -&gt; &lt; 8, 3, 22, 33, 4, 5, 4 &gt; x  -&gt; 9 y  -&gt; 8 </pre>

Now we may define sequences of arrays:

Algorithm	Output
<pre> 1 = &lt; [1, 2, 3], [4, 5] &gt;; a = 1.at(1); 1.pushBack(a); </pre>	<pre> a  -&gt; [ 4, 5 ] 1  -&gt; &lt; ([ 1, 2, 3 ]), [ 4, 5 ], [ 4, 5 ] &gt; </pre>

and arrays of structures:

Algorithm	Output
<pre> a = [ &lt; 1, 2, 3 &gt;, &lt; 4, 5, 6 &gt;, &lt; 7, 8 &gt; ]; b = a[1]; a[1].update(1, 11); a[0].insert(2,22); a[2] = a[1].pushBack(77); </pre>	<pre> a  -&gt; [ (&lt; 1, 2, 22, 3 &gt;), (&lt; 4, 11, 6 &gt;), (&lt; 4, 11, 6, 77 &gt;) ] b  -&gt; &lt; 4, 5, 6 &gt; </pre>

### 2.2.1 Structures

A structure value is of the form  $\{f_1 \rightarrow v_1 \dots f_n \rightarrow v_n\}$ , where  $f_i$  is a field name and  $v_i$  is a value, for  $i = 1, \dots, n$ .

Example:

Algorithm	Output
<pre> s = { x -&gt; 12 y -&gt; 45 }; a = s.x; s.y = 99; b.x = 22; </pre>	<pre> a  -&gt; 12 b  -&gt; { x -&gt; 22 } s  -&gt; { (x -&gt; 12) (y -&gt; 99) } </pre>

Note that the structure **b** has been created with only one field, because there is no information about its type, which is deduced on the fly during the execution.

We may have structures of arrays

Algorithm	Output
<pre> s = { x -&gt; [ 1, 2, 3 ] y -&gt; [ 4, 5, 6 ] }; b = s.y; (s.x)[1] = 11; </pre>	<pre> b  -&gt; [ 4, 5, 6 ] s  -&gt; { (x -&gt; ([ 1, 11, 3 ])) (y -&gt; [ 4, 5, 6 ]) } </pre>

sequences of arrays

Algorithm	Output
<pre> 1 = &lt; { x -&gt; 12 y -&gt; 56 }, { x -&gt; -43 y -&gt; 98 }, { x -&gt; 33 y -&gt; 66 } &gt;; u = 1.topFront(); 1.pushBack({ x -&gt; -100 y -&gt; 200 }); 1.at(2).x = 999; </pre>	<pre> a  -&gt; [ 4, 5 ] 1  -&gt; &lt; ([ 1, 2, 3 ]), [ 4, 5 ], [ 4, 5 ] &gt; </pre>

and so on.

### 2.2.2 Sets

A set value is written as  $\{v_0, \dots, v_{n-1}\}$ , where  $v_i$  is a value, for  $i = 0, \dots, n - 1$ . The operations over sets include the union  $\cup$ , the intersection  $\cap$ , the difference  $\setminus$ , and the membership test  $\_ \text{ belongsTo } \_$ . Example:

Algorithm	Output
<pre>s1 = { 1 .. 5 }; s2 = { 2, 4, 6, 7 }; a = s1 U s2 ; b = s1 ^ s2; c = s1 \ s2; t = 2 belongsTo b ^ c;</pre>	<pre>a  -&gt; { 7, 6, 1, 2, 3, 4, 5 } b  -&gt; { 2, 4 } c  -&gt; { 1, 3, 5 } s1  -&gt; { 1, 2, 3, 4, 5 } s2  -&gt; { 2, 4, 6, 7 } t  -&gt; false</pre>

Obviously, we may have sets of arrays, sequences of sets, and so on.

### 2.2.3 Specification of values

Alk includes several sugar syntax mechanisms for specifying values in a more compact way:

Algorithm	Output
<pre>p = 3; q = 9; a = [ i   i : p .. q ]; p = 2; b = [ a[i]   i : p .. p+3 ]; l = &lt; b[i] * 2   i : p-2 .. p &gt;;</pre>	<pre>a  -&gt; [ 3, 4, 5, 6, 7, 8, 9 ] b  -&gt; [ 5, 6, 7, 8 ] l  -&gt; &lt; 10, 12, 14 &gt; p  -&gt; 2 q  -&gt; 9</pre>