Alk Primer

(Draft)

Java-Semantics Version

Dorel Lucanu

May 20, 2020

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 alghorithm 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

```
print(gcd(12, 8));
```

and execute it (”gcd.alk” is the file including the above code):[1]

[1] In this document ”alki” denotes one of the two scripts running the Alk interpretes: ”alki.bat” (for Windows platform), respectively ”alki.sh” (for linux, Mac OS).

```
> alki -a gcd.alk
    4
```

An alternative is to write a general call of the algorithm

```
print(gcd(u, v));
```

and mention the initial values of the global variables `u` and `v` in the command line

```
> alki -a gcd.alk -i "u|->28v|->35"
7
```

or in an input file, say "gcd.in":

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

and give it as a parameter of the command line:

```
> alki -a gcd.alk -i gcd.in
 14
```

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

```
dfsRec(D, 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(D, j, S);
    }
  }
}

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

To execute the above algorithm on the digraph:

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

create a file "dfs.in" with the following contents:

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

and then execute the algorithm with this input:

```
> alki -a dfsrec.alk -i dfs.in
[1, 1, 1]
```

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

Chapter 2

Language Description

The examples used in this manual can be found in the folder ”doc/examples-from-manual”.

2.1 Variables and their 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 an output as expected:

```
> alki -a scalars.alk
234
true
21.468
john
```

2.1.2 Arrays

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

Algorithm	Output

`a = [3, 5, 6, 4];`
`i = 1;`
`x = a[i];`
`a[i+1] = x;`
`print(x);`
`print(a);`
	`> alki -a arrays.alk`
	`5`
	`[3, 5, 5, 4]`

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;
print(a); // [[4, 5, 6], [4, 89, 6]]
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;
print(x); // [[5, 6], [7, 8]]
print(y); // [5, 6]
print(z); // 6
print(w); // [[[1, 2], [99, 4]], [[5, 6], [7, 8]]]
```

The output is indeed the expected one:

```
>  alki -a arraysofarrays.alk
[[4, 5, 6], [4, 89, 6]]
[[5, 6], [7, 8]]
[5, 6]
6
[[[1, 2], [99, 4]], [[5, 6], [7, 8]]]
```

2.1.3 Sequences (linear lists)

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

`emptyList()`	returns the empty list $\langle\,\rangle$
L`.topFront()`	returns v_0
L`.topBack()`	returns v_{n-1}
L`.at(`i`)`	returns v_i
L`.insert(`i`,`x`)`	returns $\langle \ldots v_{i-1}, x, v_i, \ldots \rangle$
L`.removeAt(`i`)`	returns $\langle \ldots v_{i-1}, v_{i+1}, \ldots \rangle$
L`.removeAllEqTo(`x`)`	returns L, where all elements v_i equal to x were removed
L`.size()`	returns n
L`.popFront()`	returns $\langle v_1, \ldots, v_{n-1} \rangle$
L`.popBack()`	returns $\langle v_0, \ldots, v_{n-2} \rangle$
L`.pushFront(`x`)`	returns $\langle x, v_0, \ldots, v_{n-1} \rangle$
L`.pushBack(`x`)`	întoarce $\langle v_0, \ldots, v_{n-1}, x \rangle$
L`.update(`i`,`x`)`	returns $\langle \ldots v_{i-1}, x, v_{i+1}, \ldots \rangle$

Example:

Algorithm	Output

`l1 = < 8, 3, 9, 4, 5, 4 >;`
`i = 1;`
`x = l1.at(i + 1);`
`y = l1.topFront();`
`print(x);  // 9`
`print(y);  // 8`
`l1.insert(2, 22);`
`l1.update(3, 33);`
`print(l1); // < 8, 3, 22, 33, 4, 5, 4 >`
`l2 = l1;`
`l2.removeAt(0);`
`l2.removeAt(3);`
`print(l2); // < 3, 22, 33, 5, 4 >`
`l2.removeAllEqTo(4);`
`print(l2); // < 3, 22, 33, 5 >`
	`> alki -a seq.alk`
	`9`
	`8`
	`[8, 3, 22, 33, 4, 5, 4]`
	`[3, 22, 33, 5, 4]`
	`[3, 22, 33, 5]`

Now we may define sequences of arrays:

Algorithm	Output

`l = < [1, 2, 3], [4, 5] >;`
`a = l.at(1);`
`l.pushBack(a);`
`print(a);  // [4, 5]`
`print(l);  // <[1, 2, 3], [4, 5], [4, 5]>`
	`> alki -a sequencesofarrays.alk`
	`[ 4, 5 ]`
	`< [ 1, 2, 3 ], [ 4, 5 ], [ 4, 5 ] >`

and arrays of structures:

Algorithm

```
a = [ { x -> 1 y -> 2 }, { x -> 4 y -> 5 }  ];
b = a[1];
c = a[1].y;
a[1].x = 77;
print(a); // [{x -> 1, y -> 2}, {x -> 77, y -> 5}]
print(b); // {x -> 4, y -> 5}
print(c); // 5
```

Output

```
> alki -a arraysofstructures.alk
[{x -> 1, y -> 2}, {x -> 77, y -> 5}]
{x -> 4, y -> 5}
5
```

2.1.4 Structures

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

Example:

Algorithm	Output

`s = { x -> 12  y -> 45 };`
`a = s.x;`
`s.y = 99;`
`b.x = 22;`
`print(s); // {x -> 12, y -> 99}`
`print(b); // {x -> 22}`
	`> alki -a structures.alk`
	`{ (x -> 12) (y -> 99) }`
	`{ x -> 22 }`

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

```
s = { x -> [ 1, 2, 3 ] y -> [ 4, 5, 6 ] };
b = s.y;
s.x[1] = 11;
print(b); // [4, 5, 6]
print(s);  // {x -> [1, 11, 3], y -> [4, 5, 6]}
```

Output

```
> alki -a structuresofarrays.alk
[ 4, 5, 6 ]
{ (x -> ([ 1, 11, 3 ])) (y -> [ 4, 5, 6 ]) }
```

sequences of structures

Algorithm

```
l = < { x -> 12 y -> 56 }, { x -> -43 y -> 98 }, { x -> 33 y -> 66 } >;
u = l.topFront();
l.pushBack({ x -> -100 y -> 200 });
print(u);
print(l);
```

Output

```
> alki -a seqofstructures.alk
{x -> 12, y -> 56}
<{x -> 12, y -> 56}, {x -> -43, y -> 98}, {x -> 33, y -> 66}, {x -> -100, y -> 200}>
```

and so on.

2.1.5 Sets

A set value is written as $\{v_0, \ldots, v_{n-1}\}$, where v_i is a value, for $i = 0, \ldots, n-1$. The operations over sets include the union `U`, the intersection `^`, the difference `\`, and the membership test `_ in _`. Example:

Algorithm

```
s1 = { 1 .. 5 };
s2 = { 2, 4, 6, 7 };
a = s1 U s2 ;
b = s1 ^ s2;
c = s1 \ s2;
print(a); // {1, 2, 3, 4, 5, 6, 7}
print(b); // {2, 4}
print(c); // {1, 3, 5}
t = 2 in b ^ c;
print(t); // false
x = 0;
foreach y in s2 x = x + y;
print(x); // 19
d = emptySet;
foreach y in { 1 .. 6 }
  if (y in s2) d = d U singletonSet(y);
print(d); // {7}
```

Output

```
> alki -a sets.alk
{1, 2, 3, 4, 5, 6, 7}
{2, 4}
{1, 3, 5}
false
19
{2, 4, 6}
```

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

Remark. *The current implementation does check if a set value assigned to a variable is indeed a set. But the operations returns sets whenever the arguments are sets.*

2.1.6 Specification of values

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

Algorithm

```
p = 3;
q = 9;
a = [ i | i from [p .. q] ];
p = 2;
b = [ a[i] | i from [p .. p+3] ];
l = < b[i] * 2 | i from [p-2 .. p] >;
print(a);
print(b);
print(l);
```

Output

```
> alki -a specs.alk
[3, 4, 5, 6, 7, 8, 9]
[5, 6, 7, 8]
<10, 12, 14>
```

2.1.7 Lists with iterators

An iterator p associated with a list L if p "refers" an element of L. With iterators one can call operation over the associated lists and/or traverse the associated list.

Operations with ieterators:

p `+` i – returns an iterator refering the ith element after p;

p `-` i – returns an iterator refering the ith element in front of p;

`++`p – moves p to the previous element (if any);

`--`p – moves p to the next element (if any);

L`.first()` – returns an iterator that refers the first element of L;

L`.end()` – returns an invalid iterator for L

Using iterators, one can access the lements of the lists and/or execute operations on the associated list:

`*`p – returns the referred element;

p`->delete()` – remove the element referred by p and move p to the next element;

p`->insert(x)` – insert x immediately after the element referred by p.

If p refers the ith element in L, then p`->delete()` is equivalent to $L[i]$`.delete()` and p`->insert(x)` with $L[i]$`.insert(x)`.

The following operators are useful to traverse circular lists:

p `+%` i – returns an iterator refering the ith element after p modulo the length of the list;

p `-%` i – returns an iterator refering the ith element in front of p modulo the length of the list;

`++%`p – moves p to the previous element modulo the length of the list;

`--%`p – moves p to the next element modulo the length of the list

Example 1:

Algorithm

```
L = < 2, 5, 8 >;
i = 0;
A = [];
for (p = L.first(); p != L.end(); ++p) {
  A[i] = *p ;
  ++i;
}
print(A);
```

Output

```
> alki -a it1.alk
[2, 5, 8]
```

Example 2:

Algorithm

```
L = < 2, 5, 8, 33 >;
p = L.first();
p = p + 3;
a = *p;
++% p;
b = *p;
q = L.first();
c = *(--% q);
print(a);
print(b);
print(c);
```

Output

```
> alki -a it3.alk
33
2
33
```

2.2 Expressions

Alk includes the basic operators over scalars with a C++-like syntax.

Since Alk is designed with K Framework, it can be easily extended with new operators.

2.3 Statements

The syntax for the statements is similar to that of imperative C++.

We already have seen examples of the assignment statement. The other statements include:

2.3.1 Block

Syntax: `{` *Stmt* `}`

2.3.2 `if`

Syntax: 1) `if (` *Exp* `)` *Stmt* `else` *Stmt*   2) `if (` *Exp* `)` *Stmt*

2.3.3 `while`

Syntax: `while (` *Exp* `)` *Stmt*

2.3.4 `for`

Syntax: 1) `foreach` *Id* `in` *Exp* *Stmt*   2) `for (` *VarAssign* `;` *Exp* `;` *VarUpdate* `)` *Stmt*

Examples:

```
for (i= 2; i <= x / 2; ++i)
  if (x % i == 0) return false;

foreach y in { 1 .. 6 }
  if (y in s2) d = d U singletonSet(y);
```

2.3.5 Sequential Composition

Syntax: *Stmt Stmt*

2.4 Statements for Nondeterministic Algorithms

2.4.1 `choose`

Syntax: `choose` *Id* `in` *Exp* `;`   2) `choose` Id `in` *Exp* `s.t.` *Exp* `;`

Example 1:

Algorithm	Output
`choose x1 in { 1 .. 5 };` `choose x2 in { 1 .. 5 };`	`x1 |-> 3` `x2 |-> 1`

Example 2:

Algorithm	Output

`odd(x) {`
`  return x % 2 == 1;`
`}`
 	
`choose x1 in { 1 .. 5 } s.t. odd(x1);`
`choose x2 in { 1 .. 5 } s.t. odd(x2);`
	`x1 |-> 5`
	`x2 |-> 3`

Example 3:

Algorithm	Output
`choose x8 in { 1 .. 5 } s.t. x8 > 6;`	`Error at line 14: Choose can't find any suitable value.`

2.4.2 `success`

Syntax: `success ;`

Example:

Algorithm	Output

<pre>odd(x) {
  return x % 2 == 1;
}
 	
choose x in { 1 .. 8 };
if (odd(x)) success;</pre>
	<pre>> alki -a success.alk
	success
	x |-> 5</pre>

2.4.3 `failure`

Syntax: `failure ;`

Example:

Algorithm	Output

<pre>odd(x) {
  return x % 2 == 1;
}
 	
choose x in { 1 .. 8 };
if (odd(x)) success;
else failure;</pre>
	<pre>> alki -a success.alk
	failure
	x |-> 8</pre>

2.5 Functions/Procedures Describing Algorithms

Example:

Algorithm	Output ”

<pre>swap(out a, i, j) {
  temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
 	
partition(out a, p, q) {
  x = a[p] ;
  i = p + 1;   j = q;
  while (i <= j) {
    if (a[i] <= x) i = i+1;
    else if (a[j] >= x) j = j-1;
    else if (a[i] > x && x > a[j]) {
      swap(a, i, j);
      i = i+1;
      j = j-1;
    }
  }
  k = i-1; a[p] = a[k]; a[k] = x;
// if (k == q) --k;
  return k;
}
 	
qsort(out a, p, q) {
  if (p < q) {
    k = partition(a, p, q);
    qsort(a, p, k-1);
    qsort(a, k+1, q);
  }
}
 	
b = [5,1,3,2,4];
n = 5;
qsort(b, 0, n-1);
print(b);</pre>
	<pre>> alki -a qsort.alk
	[1, 2, 3, 4, 5]</pre>

Note that the output parameters and the input/output parameters are declared with the prefix `out`.

If a function modifies global variables, then these must be specified in a ”modifies” clause. Example:

Algorithm	Output”

<pre>x = 3;
y = 5;
g(b) modifies x, y {
  x = x + b;
  y = y * b;
  return x;
}
g(5);
print(x); // 8
print(y); // 25</pre>
	<pre>> alki -a globals.alk
	8
	25</pre>