Outline

Contents

1 Introducere

Ce este un algoritm?

Nu există o definiţie standard pentru noţiunea de algorithm. Putem realiza aceasta prin citirea definiţiei pentru algoritm în diverse surse considerate cu grad mare de încredere.

Cambridge Dictionary:
"A set of mathematical instructions that must be followed in a fixed order, and that, especially if given to a computer, will help to calculate an answer to a *mathematical problem*."

Schneider and Gersting 1995 (Invitation for Computer Science):
"An algorithm is a well-ordered collection of *unambiguous and effectively computable operations* that when executed produces *a result and halts in a finite amount of time*."

Gersting and Schneider 2012 (Invitation for Computer Science, 6nd edition):
"An algorithm is an ordered sequence of instructions that is *guaranteed* to solve a specific problem."

Ce este un algoritm?

Wikipedia:

"In mathematics and computer science, an algorithm is a step-by-step procedure for calculations. Algorithms are used for *calculation, data processing, and automated reasoning.*
An algorithm is an effective method expressed as a *finite list* of *well-defined instructions* for calculating a function. Starting from an *initial state* and *initial input* (perhaps empty), the instructions describe a computation that, when executed, proceeds through a *finite number of well-defined successive states*, eventually producing *"output"* and terminating at a *final ending state.* The *transition* from one state to the next is not necessarily *deterministic*; some algorithms, known as *randomized algorithms*, incorporate random input."

Ingredientele de bază: model de calcul, problemă rezolvată

Toate aceste definiţii au ceva în comun:

- datele/informaţia şi procesarea acestora/acesteia în paşi. Acestea sunt descrise în general de un model de calcul.
  Un model de calcul este format din:
  - memorie - modul de reprezentare a datelor
  - instrucţiuni
    - sintaxă - descrie sintactic paşii de procesare
    - semantică - descrie paşii de procesare realizaţi de execuţia unei instrucţiuni; în general este dată de o relaţie de tranziţie peste configuraţii (sistem tranziţional)
- un algoritm trebuie să producă un rezultat, adică un algoritm trebuie să *rezolve o problemă*.
  O problemă este în general reprezentată de o pereche *(input,output)*, unde input reprezintă descrierea datelor de intrare (instanţa) iar output descrierea datelor de ieşire (rezultatul).

Cum descriem un algoritm?

Există o varietate largă de moduri în care poate fi descris un algoritm:

- informal: limbaj natural
- formal
  - notaţie matematică (maşini Turing, lambda-calcul (Church), funcţii recursive, ...)
  - limbaje de programare: imperative de nivel inalt, de nivel jos, declarative (e.g., programare funcţională, ~~programare logică~~). Acesta poate fi şi un model informal dacă nu există o semantică formală pentru limbaj.
- semiformal
  - pseudo-cod: combină notaţia formală a limbajelor de prgramare cu limbajul natural
  - notaţie grafică: scheme logice, automate (state machines), diagrame de activităţi

De ce este nevoie de formalizare?

Următoarea poveste are scopul de a motiva de ce este nevoie de o definiţie formală pentru algoritmi şi, implicit, de ce limbajul formal sau notaţiile semiformale nu sunt suficiente.

- înainte de secolul 20 matematicienii au utilizat noţiunea de algoritm doar la nivel intuitiv
- în 1900, David Hilbert, la Congresul matematicienilor din Paris, a formulat 23 de probleme ca "provocări ale secolului"

- problema a 10-a cerea "găsirea unui proces care să determine dacă un polinom cu coeficienţi întregi are o rădăcină întreagă"
- Hilbert nu a pronunţat termenul de algoritm

De ce nevoie de formalizare?

- problema a 10-a a lui Hilbert este *nerezolvabilă*, i.e. nu există un algoritm care având la intrare un polinom p, decide dacă p are rădăcină întreagă
- acest fapt nu se poate demonstra având doar noţiunea intuitivă de algoritm
- pentru a demonstra că nu există algoritm care rezolvă o problemă, este necesară o noţiune formală

Formalizarea noţiunii de algoritm

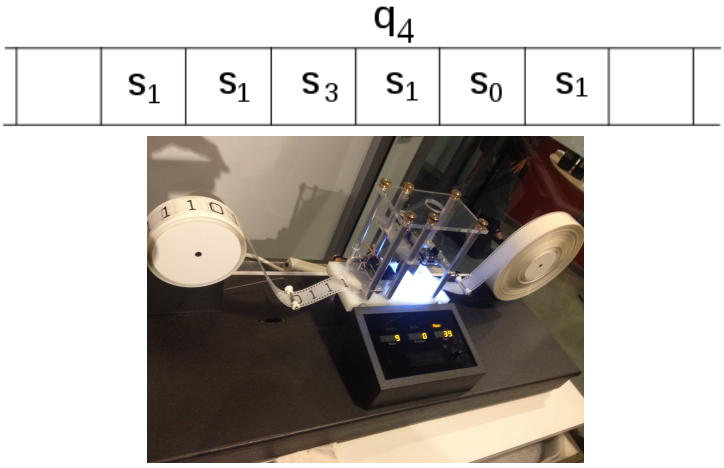
Deşi unii algoritmi îşi au originea în antichitate, de exemplu algoritmul lui Euclid, noţiunea formală de algoritm a apărut mult mai târziu.

- abia în 1936 Alonso Church şi Alan Turing au definit formal, fiecare independent, noţiunea de de algoritm
- Church a utilizat notaţia cunoscută acum sub numele de lambda-calcul (λ-calcul)
- Turing a definit modelul de calcul care acum este cunoscut sub numele de "maşini Turing"
- cele două modele de calcul sunt echivalente
- de atunci au apărut multe alte modele echivalente cu maşinile Turing
- în 1970 Yuri Matijasevic a arătat că nu există algoritm care să testeze dacă un polinom dat are rădăcină întreagă sau nu (rezultatul se bazează pe munca anterioară depusă de Martin Davis, Hilary Putnam, şi Julia Robinson), adică problema a 10-a a lui Hibert este nerezolvabilă

λ-calculus

- The language: x $\lambda x.M$ (abstraction) $M\,N$ (application)
- Booleans: $\mathit{true} \triangleq \lambda a.\lambda b.a$ $\mathit{false} \triangleq \lambda a.\lambda b.b$
- Integers: $0 \triangleq \lambda f.\lambda x.x$ (equivalent to false) $1 \triangleq \lambda f.\lambda x.f\,x$ $1 \triangleq \lambda f.\lambda x.f\,x$ $2 \triangleq \lambda f.\lambda x.f(f\,x)$ ... $\mathit{succ} = \lambda n.\lambda f.\lambda x.f(n\,f\,x)$
- Operational Semantics: $(\lambda x.M)N \Rightarrow M[N/x]$ (β-reduction)

Maşini Turing



Source:By GabrielF - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=26270095

Teza Church-Turing

Turing a afirmat că orice funcţie a cărei valoare poate fi obţinută printr-o metodă efectivă (?), poate fi calculată de o maşină Turing. (Teza lui Turing)

Forma cea mai accesibilă formulată de Turing:
⟨⟨LCMs [logical computing machines: Turing's expression for Turing machines] can do anything that could be described as "rule of thumb" or "purely mechanical".⟩⟩ (Turing 1948)

Teza lui Church:
⟨⟨A function of positive integers is effectively calculable only if recursive.⟩⟩

Modelul de calcul al funcţiilor recursive a fost definit de de Gödel şi Herbrand (1932-1934) şi este echivalent cu lambda-calculul (Kleene, Church, 1936).

Kleene (1967) este cel care a introdus termenul de Teza Church-Turing: "So Turing's and Church's theses are equivalent. We shall usually refer to them both as Church's thesis, or in connection with that one of its ... versions which deals with Turing machines as the Church-Turing thesis."

Nivelul de formalizare

Care este cel mai potrivit limbaj formal de prezentare a algoritmilor?

- maşinile Turing, lambda-calculul, funcţiile recursive au avantajul ca sunt uşor de definit matematic, dar sunt greu de utilizat în practică
- limbajele de programare sunt uşor de utilizat în practică dar dificil de manevrat în demonstraţii datorită detaliilor de implementare
- cel mai simplu limbaj de programare echivalent cu maşinile Turing:
  modelul de memorie: mulţime de variabile ce pot lua valori întregi
  instrucţiuni: `goto` condiţională şi necondiţională, incrementarea şi decrementarea variabilelor
  este cunoscut sub numele de counting machines

- o variantă structurată (modelul programelor `while`):
  - modelul de memorie: mulţime de variabile ce pot lua valori întregi
  - instrucţiuni: atribuirea, `if`, `while`, şi compunerea secvenţială

2 Limbajul Alk

Motivaţie

Scopul nostru este de a avea un limbaj care este

- destul de simplu pentru a fi înţeles;
- suficient de expresiv pentru a descrie toţi algoritmii prezentaţi;
- abstract: descrierea algoritmului face abstracţie de detaliile de implementare (specifice unui limbaj de programare real), avantajând astfel focalizarea pe gândirea algoritmică;
- să ofere un model de calcul riguros definit şi potrivit pentru analiza algoritmilor, de exemplu să ofere o bază pentru dovedirea corectitudinii unui algoritm şi pentru analiza eficienţei acestuia;
- executabil: algoritmii pot testaţi şi execuţiile lor analizate cu diverse mijloace (execuţie pas cu pas, execuţie simbolică, etc);
- intrările şi ieşirile sunt reprezentate abstract, ca obiecte matematice, astfel încât detaliile de implementare sunt omise total.

Un candidat care satisface aceste cerinţe este Alk (limbaj specific acestui curs).

2.1 Modelul de memorie

Modelul de memorie

- memoria este dată de o mulţime de variabile
- variabilele nu sunt altceva decât locaţii de memorie care stochează valori şi care sunt accesate prin nume simbolice în loc de adrese
- astfel o variabilă este o pereche:

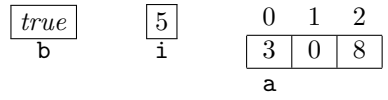
notaţie matematică	$\textit{nume-variabilă} \mapsto \textit{valoare}$
notaţie grafică	$\boxed{\textit{valoare}}$ *nume-variabilă*

- o valoare va fi o constantă a unui tip de date
- exemple de tipuri de valori:
  scalare
  tablouri
  structuri
  liste
  ...
- *Val* desemnează mulţimea tuturor valorilor

Exemple de variabile

notaţie matematică $\quad \mathtt{b} \mapsto \mathit{true} \quad \mathtt{i} \mapsto 5 \quad a \mapsto [3, 0, 8]$

notaţie grafică



Fiecare notaţie este scrierea simplificată (fără numele intermediar al funcţiei) al unei funcţii $\sigma : \{\mathtt{b}, \mathtt{i}, \ldots\} \to \mathit{Val}$ dată prin $\sigma(\mathtt{b}) = \mathit{true}$, $\sigma(\mathtt{i}) = 5$, ...

2.2 Valori

Dimensiunea valorilor

Tip de dată = valori (constante) + operaţii

Fiecare valoare (constantă) este reprezentată utilizând un spaţiu de memorie.

Pentru valorile (obiectele) fiecărui tip trebuie precizată lungimea (dimensiunea) reprezentării obiectelor.

Există două moduri de a defini lungimea reprezentării:

- uniform: $|v|_{\text{unif}}$
- logaritmic: $|v|_{\log}$

Scalari

Această categorie include tipurile primitive: valorile boolene, numerele întregi, numerele raţionale (virgulă mobilă), şiruri,...

O caracteristică importantă pentru valorile scalare: *admit reprezentări finite.*

Întrebare: ce se poate spune despre numerele iraţionale, de exemplu $\sqrt{2}$?

Scalari (cont)

- întregi:
  $\mathit{Int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - dimensiunea uniformă: $|n|_{\text{unif}} = 1$
  - dimensiunea logaritmică: $|n|_{\log} = \log_2 n$
- booleeni:
  $\mathit{Bool} = \{\mathit{false}, \mathit{true}\}$
  - dimensiunea uniformă: $|b|_{\text{unif}} = 1$
  - dimensiunea logaritmică: $|b|_{\log} = 1$
- numere raţionale:
  $\mathit{Float} =$ numerele raţionale

– dimensiunea uniformă: $|v|_{\text{unif}} = 1$

– dimensiunea logaritmică: $|v|_{\log} = \log_2(\text{mantisă}) + \log_2(\text{exponent})$

- ...

Avem $\mathit{Int} \cup \mathit{Bool} \cup \mathit{Float} \cup \ldots \subseteq \mathit{Val}$.

Tablouri (unidimensionale)

Un valoare de tip tablou (unidimensional) de lungime n poate fi reprezentată printr-o secvenţă $a = [a_0, a_1, \ldots, a_{n-1}]$ şi poate fi gândită ca o notaţie pentru funcţia $\{0 \mapsto a_0\ 1 \mapsto a_1\ \ldots\ n{-}1 \mapsto a_{n-1}\}$.

Dimensiunea unui tablou: $|a|_d = |a_0|_d + |a_1|_d + \cdots + |a_{n-1}|_d$, $d \in \{\text{unif}, \log\}$

Mulţimea tablourilor omogene de tip V este definită prin
$\mathit{Arr}_n\langle V\rangle = \{[v_0, v_1, \ldots, v_{n-1}] \mid v_i \in V, i = 0, \ldots, n{-}1\}$

Exemple: $\mathit{Arr}_n\langle \mathit{Int}\rangle$, $\mathit{Arr}_n\langle \mathit{Bool}\rangle$, $\mathit{Arr}_n\langle \mathit{Arr}_m\langle \mathit{Int}\rangle\rangle$, $\mathit{Arr}_n\langle \mathit{List}\rangle$ ...

Avem $\bigcup_{n\geq 1} \mathit{Arr}_n\langle V\rangle \subset \mathit{Val}$ pentru fiecare tip $V \subset \mathit{Val}$.

Rezultă că tablourile bidimensionale pot fi definite ca tablouri unidimensionale de tablouri unidimensionale, tablourile tridimensionale ca tablouri unidimensionale de tablouri bidimensionale, etc.

Structuri

O valoare de tip structură poate fi reprezentată ca o funcţie de la mulţimea numelor de câmpuri la cea de valori.
De exemplu, o structură `p` reprezentând puncte în plan are două câmpuri, `x` şi `y`, caz în care punctul de coordonate $(2, 7)$ este reprezentat de structura

$$\{\texttt{x} \to 2\ \texttt{y} \to 7\}.$$

Considerăm că mulţimea de câmpuri este $F = \{f_1, \ldots, f_n\}$.
O valoare de tip structură va fi în acest caz $s = \{f_1 \to v_1, \ldots, f_n \to v_n\}$.

Dimensiunea unui structuri: $|s|_d = |v_0|_d + |v_1|_d + \cdots + |v_{n-1}|_d$, $d \in \{\text{unif}, \log\}$

Mulţimea valorilor de tip structură peste F în care f_i ia valori de tip V_i este
$\mathit{Str}\langle f_1{:}V_1, \ldots f_n{:}V_n\rangle = \{\{f_1 \to v_1, \ldots, f_n \to v_n\} \mid v_1 \in V_1, \ldots, f_n \in V_n\}$

Exemplu (Fixed Size Linear Lists):
$\mathit{FSLL} = \{\texttt{len}, \texttt{arr}\}$
$\mathit{Str}\langle \texttt{len} : \mathit{Int}, \texttt{arr} : \mathit{Arr}_{100}\langle \mathit{Int}\rangle\rangle =$
$\{\{\texttt{len} \to n\ \texttt{arr} \to a\} \mid n \in \mathit{Int}, a \in \mathit{Arr}_{100}\langle \mathit{Int}\rangle\}$

Avem $\mathit{Str}\langle f_1{:}V_1, \ldots f_n{:}V_n\rangle \subset \mathit{Val}$ pentru fiecare structură $F = \{f_1{:}V_1, \ldots f_n{:}V_n\}$.

Liste liniare

O valoare de tip listă liniară este o secvenţă de valori $l = \langle v_0, v_1, \ldots, v_{n-1} \rangle$.

Dimensiunea unui structuri: $|l|_d = |v_0|_d + |v_1|_d + \cdots + |v_{n-1}|_d$, $d \in \{\text{unif}, \log\}$

$LLin\langle V \rangle = \{\langle v_0, \ldots, v_{n-1} \rangle \mid v_i \in V, i = 0, \ldots, n\}$

Exemple: $LLin\langle Int \rangle$, $LLin\langle Arr_n \rangle$, $LLin\langle Arr_n \langle Float \rangle \rangle$

Avem $LLin\langle V \rangle \subset Val$ pentru fiecare tip V.

Valori complexe: grafuri

Graful $G = (\{0, 1, 2, 3\}, \{(0, 1), (0, 2), (0, 3), (1, 2)\})$ este reprezentat prin liste de adiacenţă de următoarea valoare:

$$\begin{aligned} &\{ \\ &\texttt{n} \to 4 \\ &\texttt{a} \to [\langle 1, 2, 3 \rangle, \langle 0, 2 \rangle, \langle 0, 1 \rangle, \langle 0 \rangle] \\ &\} \end{aligned}$$

2.3 Operaţii

Tip de date (cont.)

Tip de dată = constante + operaţii

Fiecare operaţie *op* are un cost timp $time(op)$.

Pentru fiecare operaţie a unui tip trebuie precizat timpul.

Există două moduri de a defini timpul (moştenite de la lungimea reprezentării):
uniform: $time_{\text{unif}}(op)$ – utilizează dimensiunea uniformă
logaritmic: $time_{\log}(op)$ – utilizează dimensiunea logaritmică

Operaţii cu scalari

Întregi:

Operaţie	$time_{\text{unif}}(op)$	$time_{\log}(op)$
$a +_{Int} b$	$O(1)$	$O(\max(\log a, \log b))$
$a *_{Int} b$	$O(1)$	$O(\log a \cdot \log b)$ $O(\max(\log a, \log b)^{1.545})$
...	...	...

Tablouri

Operaţie	$time_{\text{unif}}(op)$	$time_{\log}(op)$
$A.\texttt{lookup}(i)$	$O(1)$	$O(\log i + \lvert a_i \rvert_{\log})$
$A.\texttt{update}(i, v)$	$O(1)$	$O(\log i + \lvert v \rvert_{\log})$

unde am presupus $A \mapsto [a_0, \ldots, a_{n-1}]$

Structuri

Operaţie	$time_{\text{unif}}(op)$	$time_{\log}(op)$
S.`lookup`(x)	$O(1)$	$O(\|s_x\|_{\log})$
S.`update`(x, v)	$O(1)$	$O(\|v\|_{\log})$

unde am presupus $S \mapsto \{\dots x \to s_x, \dots\}$

Liste liniare: definiţie operaţii

`emptyList()`	întoarce lista vidă []
L.`topFront()`	întoarce v_0
L.`topBack()`	întoarce v_{n-1}
L.`lookup(`i`)`	întoarce v_i
L.`insert(`i`,`x`)`	întoarce $[\dots v_{i-1}, x, v_i, \dots]$
L.`remove(`i`)`	întoarce $[\dots v_{i-1}, v_{i+1}, \dots]$
L.`size()`	întoarce n
L.`popFront()`	întoarce $[v_1, \dots, v_{n-1}]$
L.`popBack()`	întoarce $[v_0, \dots, v_{n-2}]$
L.`pushFront(`x`)`	întoarce $[x, v_0, \dots, v_{n-1}]$
L.`pushBack(`x`)`	întoarce $[v_0, \dots, v_{n-1}, x]$
L.`update(`i`,`x`)`	întoarce $[\dots v_{i-1}, x, v_{i+1}, \dots]$

unde am presupus $L \mapsto [v_0, \dots, v_{n-1}]$

Liste liniare: operaţii (versiunea 1)

- corespunde implementării cu tablouri

Operaţie	$time_{\text{unif}}(op)$	$time_{\log}(op)$
L.`lookup(`i`)`	$O(1)$	$O(\log i + \|v_i\|_{\log})$
L.`insert(`i, x`)`	$O(L$.`size()` $- i)$	$O(\log i + \|x\|_{\log})$
L.`remove(`i`)`	$O(L$.`size()` $- i)$	$O(\log i + \|v_i\|_{\log} + \dots + \|v_{n-1}\|_{\log}\|)$
...	...	...

Liste liniare: operaţii (versiunea 2)

- corespunde implementării cu liste dublu înlănţuite

Operaţie	$time_{\text{unif}}(op)$	$time_{\log}(op)$
L.`lookup(`i`)`	$O(i)$	$O(\log(1 + \dots + i) + \|v_i\|_{\log})$
L.`insert(`i, x`)`	$O(i)$	$O(\log(1 + \dots + i) + \|x\|_{\log})$
L.`remove(`i`)`	$O(i)$	$O(\log(1 + \dots + i))$
...	...	...

2.4 Expresii şi instrucţiuni

2.4.1 Sintaxa

Expresii: sintaxa

Sintaxa expresiilor este foarte apropiată de cea din limbajul C++. Prezentăm mai jos câteva exemple de expresii Alk:

- expresii aritmetice: `a * b + 2`
- expresii relaţionale: `a < 5`
- expresii booleene: `(a < 5) && (a > -1)`

- expresii peste mulţimi: `s1 U s2  s1 ^ s2  s1 \ s2`
- apel de funcţie: `f(a*2, b+5)`
- apel operaţii peste liste, mulţimi, tablouri: `l.update(2,55)  l.size()`

Sintaxa formală utilizând BNF (opţional):

```
Exp ::=
     Id
   | Value
   | FunctionCall
   | IncDec
   | (Exp)                    [bracket]
   | "emptyList"
   | "emptySet"
   | "sin" "(" Exp ")"
   | "cos" "(" Exp ")"
   | "tan" "(" Exp ")"
   | "asin" "(" Exp ")"
   | "acos" "(" Exp ")"
   | "atan" "(" Exp ")"
   | "sqrt" "(" Exp ")"
   | "ceil" "(" Exp ")"
   | "floor" "(" Exp ")"
   | "int" "(" Exp ")"
   | "new" "(" Exp ")"
   | "{" Exp "}"   // singleton set
   | "<" Exp ">"  // singleton list
   | "{" Interval "}"
   | "<" Interval ">"
   | "[" Interval "]"
   > Exp "[" Exp "]"
   > Exp "." "at" "(" Exp ")"
   | Exp "." "topFront" "(" ")"
   | Exp "." "topBack" "(" ")"
   | Exp "." "size" "(" ")"
   | Exp "." "update" "(" Exp "," Exp ")"
   | Exp "." "insert" "(" Exp "," Exp ")"
   | Exp "." "removeAt" "(" Exp ")"
   | Exp "." "removeAllEqTo" "(" Exp ")"
   | Exp "." "pushFront" "(" Exp ")"
   | Exp "." "pushBack" "(" Exp ")"
   | Exp "." "popFront" "(" ")"
   | Exp "." "popBack" "(" ")"
   > Exp "." Id                           [left]
   > left:
     Exp "*" Exp           [left]
   | Exp "/" Exp           [left]
   | Exp "%" Exp           [left]
   > left:
     Exp "+" Exp           [left]
   | Exp "-" Exp           [left]
   > left:
     Exp "U" Exp           [left]
   | Exp "^" Exp           [left]
   | Exp "\" Exp          [left]
   > Exp OpRel Exp         [noassoc]
   > Exp "belongsTo" Exp
   > "!" Exp
   > left:
     Exp "&&" Exp          [left]
   | Exp "||" Exp          [left]

Exps ::= List{Exp,","}

FunctionCall ::= Exp "("  Exps ")"
```

Instrucţiuni: sintaxa

Alk include un nucleu de instrucţiuni imperative a căror sintaxă este similară cu cea din limbajul C++.

- atribuire `a =` E`;` `a[i] =` E`;` `p.x =` E`;`
- apeluri de funcţii: `quicksort(a);` `l.insert(2,77);`
- bloc: `{` Sts `}`
- instrucţiuni condiţionale:
  - `if (` E `)` St
  - `if (` E `)` St_1 `else` St_2
- instrucţiuni repetitive:
  - `while (` E `)` St
  - `forall` X `in` S St
  - `for (` X `=` E`;` E'`;` `++`X `)` S
- return: `return` E`;`
- compunerea secvenţială: St_1 St_2

Sintaxa formală utilizând BNF (opţional):

```
VarAssign ::= Exp "=" Exp

     IncDec ::=
         "++" Id
       | "--" Id

     VarUpdate ::=
         VarAssign
       | IncDec

    Stmt ::=
          Id "(" Ids ")" "{" Stmt "}"
       |  FunctionCall ";"
       |  VarUpdate ";"
       |  Exp "." "update" "(" Exp "," Exp ")" ";"
       |  Exp "." "insert" "(" Exp "," Exp ")" ";"
       |  Exp "." "removeAt" "(" Exp ")" ";"
       |  Exp "." "removeAllEqTo" "(" Exp ")" ";"
       |  Exp "." "pushFront" "(" Exp ")" ";"
       |  Exp "." "pushBack" "(" Exp ")" ";"
       |  Exp "." "popFront" "(" ")" ";"
       |  Exp "." "popBack" "(" ")" ";"
       |  "{" "}"
       |  "{" Stmt "}"
       | "if" "(" Exp ")" Stmt "else" Stmt         [avoid]
```

```
        | "if" "(" Exp ")" Stmt
        |  "while" "(" Exp ")" Stmt
        |  "forall" Id "in" Exp  Stmt
        |  "for" "(" VarAssign ";" Exp";" VarUpdate ")" Stmt
        | "return" Exp ";"
        > Stmt Stmt [right]

    Stmts ::= List{Stmt,""}
```

Alk este extensibil: pot fi adăugate noi expresii şi instrucţiuni, cu precizări pentru costul timp

Tipuri de date

Sunt predefinite în Alk.

Presupunem existenţa unei metainformaţii care menţionează tipul fiecărei variabile (nu există declaraţii de variabile).

Exemplu de program

```
/*
  This example includes the recursive version of the DFS algorithm.
  @input: a digraf D and a vertex i0
  @output: the list S of the vertices reachable from i0
*/

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

```
// the calling program
i = 0;
while (i < D.n) {
  S[i] = 0;
  i = i + 1;
}
dfsRec(1);
```

În exemplul de mai sus se presupune că `D` este o structură ce reprezintă un graf prin liste de adiacenţă (definită ca mai sus), `S` este un tablou unidimensional cu elemente numere întregi, iar `p` este o listă.

2.4.2 Semantica

Evaluarea expresiilor

Notaţie: $[\![E]\!](\sigma)$ reprezintă rezultatul evaluării expresiei E în starea σ. Intuitiv, $[\![E]\!](\sigma)$ se obţine prin înlocuirea variabilelor din E cu valorile acestora din starea σ şi appoi se efctuează operaţiile. Formal, considerăm o funcţie $[\![\_]\!](\_) : \mathit{Expresii} \to (\mathit{Stare} \to \mathit{Val})$, unde unei expresii $E \in \mathit{Expresii}$ i se asociază o funcţie $(\mathit{Stare} \to \mathit{Val})$, care la rândul său întoarce valoarea lui E corespunzătoare unei stări date $\sigma \in \mathit{Stare}$.

Exemplu: Fie σ o stare ce include $\mathtt{a} \mapsto 3$ $\mathtt{b} \mapsto 6$. Avem:
$[\![\mathtt{a} + \mathtt{b} * 2]\!](\sigma) =$
$[\![\mathtt{a}]\!](\sigma) +_{Int} [\![\mathtt{b} * 2]\!](\sigma) =$
$3 +_{Int} [\![b]\!](\sigma) *_{Int} [\![2]\!](\sigma) =$

$3 +_{Int} 6 *_{Int} 2 =$
$3 +_{Int} 12 = 15$
unde $+_{Int}$ reprezintă algoritmul de adunare peste întregi şi $*_{Int}$ reprezintă algoritmul de înmulţire peste întregi.

Formal, $[\![\_]\!](\_)$ este definită prin reguli de calcul de forma (parţial):

$$\overline{[\![\mathtt{x}]\!](\ldots \mathtt{x} \mapsto \mathtt{v} \ldots) = \mathtt{v}} \qquad \overline{[\![v]\!](\_) = v}$$

$$\frac{[\![E_1]\!](\sigma) = v_1 \quad [\![E_2]\!](\sigma) = v_2}{[\![E_1 + E_2]\!](\sigma) = v_1 +_{Int} v_2}$$

...

Calculul timpului pentru evaluare

Am vaăzut că operaţiile au asociate funcţa de cost timp (necesar efectuării acestora pentru argumentele date). Funcţa de cost timp este extinsă în mod firesc la expresii: intuitiv, timpul de evaluare al unei expresii se obţine prin însumarea timpilor necesari efectuării operaţiilor care apar în expresie. $time_d([\![\mathtt{a} + \mathtt{b} * \mathtt{2}]\!](\sigma)) = time_{\mathtt{d}}([\![\mathtt{a}]\!](\sigma)) + time_{\mathtt{d}}([\![\mathtt{b}]\!](\sigma)) + time_{\mathtt{d}}(\mathtt{6} *_{Int} \mathtt{2}) + time_{\mathtt{d}}(\mathtt{3} +_{Int} \mathtt{12})$, $d \in \{\text{unif}, \log\}$.

$time_{\log}([\![\mathtt{a}]\!](\sigma)) = \log \mathtt{3}$, $time_{\log}([\![\mathtt{b}]\!](\sigma)) = \log \mathtt{6}$
$time_{\text{unif}}([\![\mathtt{a}]\!](\sigma)) = \mathtt{1}$, $time_{\text{unif}}([\![\mathtt{b}]\!](\sigma)) = \mathtt{1}$

Regulile de mai sus, care definesc evaluarea, ne dau si o metoda de calcul a timpului necesar evaluării:

$time([\![\mathtt{x}]\!](\ldots \mathtt{x} \mapsto \mathtt{v} \ldots)) = \log \mathtt{v}$
$time([\![\mathtt{x}]\!](\ldots \mathtt{x} \mapsto \mathtt{v} \ldots)) = \mathtt{1}$ *(uniform)*

$time([\![E_1 + E_2]\!](\sigma)) = time([\![E_1]\!](\sigma)) + time([\![E_2]\!](\sigma)) + time([\![E_1]\!](\sigma) +_{Int} [\![E_2]\!](\sigma))$
...

Configuraţii

O configuraţie include elementele necasare executării unui pas al algoritmului şi, în cazul modelului descris de Alk, este dată de o pereche $\langle$*secvenţă-de-program*, *stare*$\rangle$.

Exemple:
$\langle$`x = x + 1; y = y + 2 * x;`, $\mathtt{x} \mapsto 7\ \mathtt{y} \mapsto 12\rangle$

$\langle$`s = 0; while (x > 0) {s = s+x; x = x-1;}`, $\mathtt{x} \mapsto 5\ \mathtt{s} \mapsto -15\rangle$

Considerarea de funcţii presupune adaăugarea la configuraţie a unei componente suplimentare, stiva. Pentru a mentţine prezentarea cât mai simplă, definiţia formală a acesteia este omisă.

Paşi de execuţie

Un pas de execuţie este definit ca o tranziţie între configuraţii:

$$\langle S, \sigma \rangle \Rightarrow \langle S', \sigma' \rangle$$

dacă şi numai dacă

executând prima instrucţiune secvenţa din S în starea σ obţinem secvenţa S', ce urmează a fi executată în continuare, şi o nouă stare σ'.

Paşii de execuţie sunt descrişi prin reguli $\langle S_1, \sigma_1\rangle \Rightarrow \langle S_2, \sigma_2\rangle$, unde $S_1, S_2, \sigma_1, \sigma_2$ sunt termeni cu variabile (patterns).

Pentru a putea calcula costul timp al unui pas de execuţie, vom preciza costul dat de aplicarea unei reguli.

Instrucţiuni: semantica

atribuirea: `x =` E`;`

- *informal*: se evaluează E şi rezultatul este atribuit ca noua valoare a variabilei `x`
- *formal*:
  $\langle$`x =` E`;`$S, \sigma\}\rangle \Rightarrow \langle S, \sigma'\rangle$
  unde σ este de forma $\ldots$`x` $\mapsto v \ldots$ şi σ' de forma $\ldots$`x` $\mapsto [\![\mathrm{E}]\!](\sigma)\ldots$ (în rest la fel ca σ).

Costul timp:
$time_{\text{unif}}(\langle$`x =` E`;`$S, \sigma\rangle \Rightarrow \langle S, \sigma'\rangle) = time_{\text{unif}}([\![E]\!](\sigma)) + 1$,
$time_{\log}(\langle$`x =` E`;`$S, \sigma\rangle \Rightarrow \langle S, \sigma'\rangle) = time_{\log}([\![E]\!](\sigma) + \log[\![E]\!](\sigma))$.

Instrucţiuni: semantica

if: `if (`E`) then` S `else` S'

- *informal*: se evaluează e; dacă rezultatul obţinut este *true*, atunci se execută S, altfel se execută S'
- *formal*:
  $\langle$`if (`E`)` S `else` S' $S'', \sigma\rangle \Rightarrow \langle S\ S'', \sigma\rangle$ dacă $[\![E]\!](\sigma) = \mathit{true}$
  $\langle$`if (`E`)` S `else` S' $S'', \sigma\rangle \Rightarrow \langle S'\ S'', \sigma\rangle$ dacă $[\![E]\!](\sigma) = \mathit{false}$

Costul timp:
$time_d(\langle$`if (`E`)` S' `else` S'' $S, \sigma\rangle \Rightarrow \langle \_, \sigma\rangle) = time_d([\![E]\!](\sigma))$
$d \in \{\text{unif}, \log\}$.

Instrucţiuni: semantica

while: `while (`E`)` S

- *informal*: se evaluează e; dacă rezultatul obţinut este *true*, atunci se execută S, după care se evaluează din nou e şi $\ldots$; altfel execuţia instrucţiunii se termină
- *formal*: se exprimă cu ajutorul lui `if`:

  $\langle$`while (`e`)` S $S', \sigma\rangle \Rightarrow \langle$`if (`$e$`) { `$S$ `; while (`e`)` S `} else { }`$S', \sigma\rangle$

Costul timp:
$time_d(\langle$`while (E) then` S `else` S' $S, \sigma\rangle \Rightarrow \langle$`if (`$e$`) ...`$S, \sigma\rangle) = 0$,
$d \in \{\text{unif}, \log\}$.

Apelul de funcţie

Operaţie mai complexă ce include mai multe activităţi:

- salvarea stării curente în stivă
- se evaluează argumentele actuale
- legarea parametrilor formali la valorile date de argumentele actuale (parametrii funcţiei devin variabile cu valorile date de evaluarea argumentelor actuale)
- execuţia corpului funcţiei
- după execuţia corpului funcţiei, se restaurează starea de dinainte de apel actualizată cu valorile numelor globale modifcate de funcţie

Presupunem funcţia `f(a,b) { }` S_f. Evaluarea apelului `f`(e_1,e_2) presupune următorii paşi:

$\langle \texttt{f}(e_1, e_2)\ S, \sigma, \mathit{Stack}\rangle \Rightarrow$

$\langle S_f, \sigma \cup \{\texttt{a} \mapsto [\![e_1]\!](\sigma)\ \texttt{b} \mapsto [\![e_2]\!](\sigma)\}, (S, \sigma)\ \mathit{Stack}\rangle \Rightarrow^*$

$\langle v, \sigma', (S, \sigma)\ \mathit{Stack}\rangle \Rightarrow$

$\langle v\ S, \mathit{updateGlobals}(\sigma, \sigma'), \mathit{Stack}\rangle$

Presupunere: costul unui apel este suma dintre costul evaluării argumentelor actuale şi costul execuţiei corpului funcţiei.

Calcul (execuţie)

Un calcul (o execuţie) este o secvenţă de paşi: $\tau = \langle S_1, \sigma_1\rangle \Rightarrow \langle S_2, \sigma_2\rangle \Rightarrow \langle S_3, \sigma_3\rangle \Rightarrow \ldots$

Costul timp unui calcul:
$time_d(\tau) = \sum_i time_d(\langle S_i, \sigma_i\rangle \Rightarrow \langle S_{i+1}, \sigma_{i+1}\rangle)$, $d \in \{unif, log\}$

Calcul: exemplu

$\langle$`if (x > 3) x = x + y; else x = 0; y = 4;` $, \texttt{x} \mapsto 7\ \texttt{y} \mapsto 12\rangle \Rightarrow$
$\langle$`x = x + y; y = 4;` $, \texttt{x} \mapsto 7\ \texttt{y} \mapsto 12\rangle \Rightarrow$
$\langle$`y = 4;` $, \texttt{x} \mapsto 19\ \texttt{y} \mapsto 12\rangle \Rightarrow$
$\langle \cdot, \texttt{x} \mapsto 19\ \texttt{y} \mapsto 4\rangle$

Am utilizat:
$[\![\texttt{x} > 3]\!](\texttt{x} \mapsto 7\ \texttt{y} \mapsto 12) = \mathit{true}$
$[\![\texttt{x} + \texttt{y}]\!](\texttt{x} \mapsto 7\ \texttt{y} \mapsto 12) = 19$
$[\![4]\!](\texttt{x} \mapsto 19\ \texttt{y} \mapsto 12) = 4$

Costul:
cost uniform: 3 (= numărul de paşi)
cost logaritmic: $\log 7 + \log 7 + \log 12 + \log 19 + \log 4$

Calcul: exemplu

$\langle$`while (i > 5) i--;` , `i` $\mapsto 6$ `x` $\mapsto 12\rangle \Rightarrow$
$\langle$`if (i > 5) { i --; while (i > 5) i--; }`, `i` $\mapsto 1$ `x` $\mapsto 12\rangle \Rightarrow$
$\langle$`{i --; while (i > 5) i--; }` , `i` $\mapsto 6$ `x` $\mapsto 12\rangle \Rightarrow$
$\langle$`i --; while (i > 5) i--;` , `i` $\mapsto 6$ `x` $\mapsto 12\rangle \Rightarrow$
$\langle$`while (i > 5) i--;` , `i` $\mapsto 5$ `x` $\mapsto 12\rangle \Rightarrow$
$\langle \cdot$, `i` $\mapsto 5$ `x` $\mapsto 12\rangle$

Am utilizat:
$[\![$`i` $> 5]\!]$(`i` $\mapsto$ `6 x` $\mapsto$ `12`) = *true*
$[\![$`i` $- -]\!]$(`i` $\mapsto$ `6 x` $\mapsto$ `12`) = `0`
$[\![$`i` $> 5]\!]$(`i` $\mapsto$ `5 x` $\mapsto$ `12`) = `false`

Costul timp:
cost uniform: 4 (= numărul de paşi, exceptând primul)
cost logaritmic: $\log 6 + \log 6 + \log 5 + \log 5$

3 Testarea algoritmilor cu K Framework

Testarea algoritmilor cu *K Framework*

K Framework (www.kframework.org) este un mediu de lucru pentru definiţii de limbaje de programare. Definiţiile K sunt executabile. Limbajul Alk este definit în K, aşa că algoritmii descrişi în Alk pot fi testaţi şi analizaţi.

Definiţia K a lui Alk se găseşte la adresa
https://github.com/alk-language/k-semantics
şi poate fi compilată cu K versiunea 3.6:
https://github.com/kframework/k/releases
O scurtă descriere a limbajului: https://github.com/alk-language/documentation

Compilarea definiţiei limbajului Alk:
kompile alk

Proiecte (posibil pentru licenţă):
1. implementare C++ a definiţiei lui Alk;
2. dezvoltarea unei interfeţe specifice pentru definiţia lui Alk.

Execuţia algoritmului DFS recursiv

Starea iniţială este dată ca valoare a opţiunii `init`:

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

Obţinerea configuraţiei finale

Configuraţia finală obţinută prin comanda precedentă este:

```
State:

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

Componentele unei confiraţii în K sunt celule, descrise cu o sintaxă inspirată din XML. Celula `<k>` S `</k>` include lista de activităţi S de executat (în cpnfiguraţia iniţială ea include programul), iar celula `<state>` σ `</state>` include starea memoriei σ. Funcţiile program sunt memorate ca lambda-expresii.

Demo

Execuţia algoritmilor de mai sus.

Alk cu calculul timpului

```
alk-with-time$ ~/Software/k3.6/bin/kompile alk.k
alk-with-time$ ~/Software/k3.6/bin/krun tests/sum.alk -cINIT="n |-> 1000"
<k>
    .K
</k>
<state>
    n |-> 0
    s |-> 500500
</state>
<ltime>
    33384
</ltime>
<utime>
    3001
</utime>
```