

Outline

Contents

1	Introducere	1
2	Limbajul Alk	4
2.1	Modelul de memorie	4
2.2	Valori	5
2.3	Operații	7
2.4	Expresii și instrucțiuni	9
2.4.1	Sintaxa	9
2.4.2	Semantica	12
3	Testarea algoritmilor cu K Framework	15

1 Introducere

Ce este un algoritm

Nu există o definiție standard pentru noțiunea de algorithm. Putem realiza aceasta prin citirea definiției pentru algorithm 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 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 definiții au ceva în comun:

- datele/informația și procesarea acestora/acesteia în pași. Acestea sunt descrise în general de un model de calcul. Un model de calcul este format din:
 - memorie - modul de reprezentare a datelor
 - instrucțiuni
 - * sintaxă - descrie sintactic pașii de procesare
 - * semantică - descrie pașii de procesare realizați de execuția unei instrucțiuni; în general este dată de o relație de tranziție peste configurații (sistem tranzițional)
- 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 (instanța) iar output descrierea datelor de ieșire (rezultatul).

Cum descriem un algoritm?

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

- informal: limbaj natural
- formal
 - notație matematică (mașini Turing, lambda-calcul (Church), funcții recursive, ...)
 - limbaje de programare: imperative de nivel înalt, de nivel jos, declarative (e.g., programare funcțională, ~~programare logică~~). Acesta poate fi și un model informal dacă nu există o semantică formală pentru limbaj.
- semiformal
 - pseudo-cod: combină notația formală a limbajelor de programare cu limbajul natural
 - notație grafică: scheme logice, automate (state machines), diagrame de activități

De ce este nevoie de formalizare?

Următoarea poveste are scopul de a motiva de ce este nevoie de o definiție formală pentru algoritmi și, implicit, de ce limbajul formal sau notațiile semi-formale nu sunt suficiente.

- înainte de secolul 20 matematicienii au utilizat noțiunea 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 coeficienți întregi are o rădăcină întreagă"
- Hilbert nu a pronunțat 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 noțiunea intuitivă de algoritm
- pentru a demonstra că nu există algoritm care rezolvă o problemă, este necesară o noțiune formală

Formalizarea noțiunii de algoritm

Deși unii algoritmi își au originea în antichitate, de exemplu algoritmul lui Euclid, noțiunea formală de algoritm a apărut mult mai târziu.

- abia în 1936 Alonso Church și Alan Turing au definit formal, fiecare independent, noțiunea de de algoritm
- Church a utilizat notația cunoscută acum sub numele de lambda-calcul (λ -calcul)
- Turing a definit modelul de calcul care acum este cunoscut sub numele de "mașini Turing"
- cele două modele de calcul sunt echivalente
- 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, și Julia Robinson), adică problema a 10-a a lui Hilbert este nerezolvabilă
- de atunci au apărut multe alte modele echivalente cu mașinile Turing

Teza Church-Turing

Turing a afirmat că orice funcție a cărei valoare poate fi obținută printr-o metodă efectivă (?), poate fi calculată de o mașină Turing. (Teza lui Turing)

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

Teza lui Church: $\langle\langle$ A function of positive integers is effectively calculable only if recursive. $\rangle\rangle$

Modelul de calcul al funcțiilor recursive a fost definit de Gödel și Herbrand (1932-1934) și 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?

- mașinile Turing, lambda-calculul, funcțiile recursive au avantajul ca sunt ușor de definit matematic, dar sunt greu de utilizat în practică
- limbajele de programare sunt ușor de utilizat în practică dar dificil de manevrat în demonstrații datorită detaliilor de implementare
- cel mai simplu limbaj de programare echivalent cu mașinile Turing: modelul de memorie: mulțime de variabile ce pot lua valori întregi instrucțiuni: `goto` condițională și necondițională, incrementarea și decrementarea variabilelor este cunoscut sub numele de counting machines
- o variantă structurată (modelul programelor `while`):
 - modelul de memorie: mulțime de variabile ce pot lua valori întregi
 - instrucțiuni: atribuirea, `if`, `while`, și compunerea secvențială

2 Limbajul Alk

Motivație

Scopul nostru este de a avea un limbaj care este

- destul de simplu pentru a fi înțeles;
- suficient de expresiv pentru a descrie toți algoritmi prezentați;
- abstract: descrierea algoritmului face abstracție de detaliile de implementare (specifice unui limbaj de programare real), avantajând astfel focalizarea pe gândirea algoritmică
- să ofere un model de calcul riguros definit și potrivit pentru analiza algoritmilor, de exemplu să ofere o bază pentru dovedirea corectitudinii unui algoritm și pentru analiza eficienței acestuia;
- executabil: algoritmi pot testați și execuțiile lor analizate cu diverse mijloace (execuție pas cu pas, execuție simbolică, etc);
- intrările și ieșirile sunt reprezentate abstract, ca obiecte matematice, astfel încât detaliile de implementare sunt omise total.

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

2.1 Modelul de memorie

Modelul de memorie

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

notație matematică $\text{nume-variabilă} \mapsto \text{valoare}$

notație grafică $\boxed{\text{valoare}}$
 nume-variabilă

- o valoare va fi un obiect al unui tip de date
- exemple de tipuri de valori: scalare tablouri structuri liste ...
- *Val* desemnează mulțimea tuturor valorilor

Exemple de variabile

notație matematică $\mathbf{b} \mapsto \text{true}$ $\mathbf{i} \mapsto 5$ $\mathbf{a} \mapsto [3, 0, 8]$

notație grafică $\boxed{\text{true}}$ $\boxed{5}$ $\begin{array}{ccc} 0 & 1 & 2 \\ \hline 3 & 0 & 8 \end{array}$
 \mathbf{b} \mathbf{i} \mathbf{a}

Fiecare notație este scrierea simplificată (fără numele intermediar al funcției) al unei funcții $\sigma : \{\mathbf{b}, \mathbf{i}, \dots\} \rightarrow \text{Val}$ dată prin $\sigma(\mathbf{b}) = \text{true}$, $\sigma(\mathbf{i}) = 5$, ...

2.2 Valori

Dimensiunea valorilor

Tip de dată = valori (obiecte) + operații

Fiecare valoare (obiect) este reprezentată utilizând un spațiu 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 raționale (virgulă mobilă), șiruri, ...

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

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

Scalari (cont)

- întregi: $\text{Int} = \{\dots, -2, -1, 0, 1, 2, \dots\}$
 - dimensiunea uniformă: $|n|_{\text{unif}} = 1$
 - dimensiunea logaritmică: $|n|_{\log} = \log_2 n$

- booleeni: $Bool = \{false, true\}$
 - dimensiunea uniformă: $|b|_{\text{unif}} = 1$
 - dimensiunea logaritmică: $|b|_{\log} = 1$
- numere raționale: $Float =$ numerele raționale
 - dimensiunea uniformă: $|v|_{\text{unif}} = 1$
 - dimensiunea logaritmică: $|v|_{\log} = \log_2(\text{mantisă}) + \log_2(\text{exponent})$
- ...

Avem $Int \cup Bool \cup Float \cup \dots \subseteq Val$.

Tablouri (unidimensionale)

Un valoare de tip tablou (unidimensional) de lungime n poate fi reprezentată printr-o secvență $a = [a_0, a_1, \dots, a_{n-1}]$ și poate fi gândită ca o notație pentru funcția $\{0 \mapsto a_0 \ 1 \mapsto a_1 \ \dots \ n-1 \mapsto a_{n-1}\}$.

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

Mulțimea tablourilor omogene de tip V este definită prin $Arr_n\langle V \rangle = \{[v_0, v_1, \dots, v_{n-1}] \mid v_i \in V, i = 0, \dots, n-1\}$

Exemple: $Arr_n\langle Int \rangle, Arr_n\langle Bool \rangle, Arr_n\langle Arr_m\langle Int \rangle \rangle, Arr_n\langle List \rangle \dots$

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

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 funcție de la mulțimea numelor de câmpuri la cea de valori. De exemplu, o structură \mathbf{p} reprezentând puncte în plan are două câmpuri, \mathbf{x} și \mathbf{y} , caz în care punctul de coordonate $(2, 7)$ este reprezentat de structura $\{\mathbf{x} \rightarrow 2 \ \mathbf{y} \rightarrow 7\}$.

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

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

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

Exemplu (Fixed Size Linear Lists): $FSL = \{\mathbf{len}, \mathbf{arr}\} \ Str\langle \mathbf{len} : Int, \mathbf{arr} : Arr_{100}\langle Int \rangle \rangle = \{\{\mathbf{len} \rightarrow n \ \mathbf{arr} \rightarrow a\} \mid n \in Int, a \in Arr_{100}\langle Int \rangle\}$

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

Liste liniare

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

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

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

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 adiacență de următoarea valoare:

$$\begin{cases} \mathbf{n} \rightarrow 4 \\ \mathbf{a} \rightarrow [\langle 1, 2, 3 \rangle, \langle 0, 2 \rangle, \langle 0, 1 \rangle, \langle 0 \rangle] \\ \end{cases}$$

2.3 Operații

Tip de date (cont.)

Tip de dată = obiecte + operații

Fiecare operație op are un cost timp $time(op)$.

Pentru fiecare operație a unui tip trebuie precizat timpul.

Există două moduri de a defini timpul (moștenite de la lungimea reprezentării):

uniform: $time_{\text{unif}}(op)$ – utilizează dimensiunea uniformă

logaritmic: $time_{\log}(op)$ – utilizează dimensiunea logaritmică

Operații cu scalari

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

Tablouri

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

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

Structuri

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

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

Liste liniare: definiție operații

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

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

Liste liniare: operații (versiunea 1)

- corespunde implementării cu tablouri

Operație	$time_{unif}(op)$	$time_{log}(op)$
<code>L.lookup(i)</code>	$O(1)$?
<code>L.insert(i, x)</code>	$O(L.size() - i)$?
<code>L.remove(i, x)</code>	$O(L.size() - i)$?
<code>L.update(i, x)</code>	$O(1)$?
<code>L.topFront()</code>	$O(1)$?
<code>L.popFront()</code>	$O(L.size())$?
<code>L.pushFront()</code>	$O(L.size())$?
<code>L.topBack()</code>	$O(1)$?
<code>L.popBack()</code>	$O(1)$?
<code>L.pushBack()</code>	$O(1)$?

Liste liniare: operații (versiunea 2)

- corespunde implementării cu liste dublu înlanțuite

Operație	$time_{unif}(op)$	$time_{log}(op)$
<code>L.lookup(i)</code>	$O(i)$?
<code>L.insert(i, x)</code>	$O(i)$?
<code>L.remove(i, x)</code>	$O(i)$?
<code>L.update(i, x)</code>	$O(i)$?
<code>L.topFront()</code>	$O(1)$?
<code>L.popFront()</code>	$O(1)$?
<code>L.pushFront()</code>	$O(1)$?
<code>L.topBack()</code>	$O(1)$?
<code>L.popBack()</code>	$O(1)$?
<code>L.pushBack()</code>	$O(1)$?

Liste liniare: operații (versiunea 3?)

- Există o listă liniară cu următoarele proprietăți?

Operație	$time_{unif}(op)$	$time_{log}(op)$
<code>L.lookup(i)</code>	$O(1)$?
<code>L.insert(i, x)</code>	$O(i)$?
<code>L.remove(i, x)</code>	$O(i)$?
<code>L.update(i, x)</code>	$O(1)$?
<code>L.topFront()</code>	$O(1)$?
<code>L.popFront()</code>	$O(1)$?
<code>L.pushFront()</code>	$O(1)$?
<code>L.topBack()</code>	$O(1)$?
<code>L.popBack()</code>	$O(1)$?
<code>L.pushBack()</code>	$O(1)$?

2.4 Expresii și instrucțiuni

2.4.1 Sintaxa

Expresii: sintaxa

Sintaxa este foarte apropiată de cea limbajului C++:

- expresii aritmetice: `a * b + 2`
- expresii relaționale: `a < 5`
- expresii booleene: `(a < 5) && (a > -1)`
- expresii peste mulțimi: `s1 U s2` `s1 ^ s2` `s1 \ s2`
- apel de funcție: `f(a*2, b+5)`
- apel operații peste liste, mulțimi, tablouri: `l.update(2,55)` `l.size()`

Sintaxa formală utilizând BNF:

Notăția Backus-Naur

```

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 Varianta recursivă
  | "<" 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:
    Exp "*" Exp
    | Exp "/" Exp
    | Exp "%" Exp
  > left:
    Exp "+" Exp
    | Exp "-" Exp
  > left:
    Exp "U" Exp
    | Exp "^" Exp
    | Exp "\" Exp
  > Exp OpRel Exp
  > Exp "belongsTo" Exp
  > "!" Exp
  > left:
    Exp "&&" Exp
    | Exp "||" Exp

```

$$\text{Exps} ::= \text{List}\{\text{Exp}, ", "\}$$

Instrucțiuni: sintaxa

- atribuire $a = E$; $a[i] = E$; $p.x = E$;
- apeluri de funcții: `quicksort(a)`; `l.insert(2,77)`;
- bloc: $\{ Sts \}$
- instrucțiuni condiționale:
 - `if (E) St`
 - `if (E) St1 else St2`
- instrucțiuni repetitive:
 - `while (E) St`
 - `forall X in S St`
 - `for (X = E; E'; ++X) S`
- return: `return E`;
- compunerea secvențială: $St_1 St_2$

Sintaxa formală utilizând BNF:

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

`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 și instrucțiuni, cu precizări pentru costul timp

Tipuri de date

Sunt predefinite în Alk.

Presupunem existența unei metainformații care menționează tipul fiecărei variabile (nu există declarații 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);

```

2.4.2 Semantica

Evaluarea expresiilor

Considerăm o funcție $\llbracket \cdot \rrbracket (-) : \text{Expresii} \rightarrow (\text{Stare} \rightarrow \text{Valori})$, unde $\llbracket E \rrbracket (\sigma)$ întoarce valoarea expresiei E calculată în starea σ .

Exemplu: Fie σ o stare ce include $\mathbf{a} \mapsto 3$ $\mathbf{b} \mapsto 6$. Avem: $\llbracket \mathbf{a} + \mathbf{b} * 2 \rrbracket (\sigma) = \llbracket \mathbf{a} \rrbracket (\sigma) +_{Int} \llbracket \mathbf{b} * 2 \rrbracket (\sigma) = 3 +_{Int} \llbracket \mathbf{b} \rrbracket (\sigma) *_{Int} \llbracket 2 \rrbracket (\sigma) = 3 +_{Int} 6 *_{Int} 2 = 3 +_{Int} 12 = 15$ unde $+_{Int}$ reprezintă algoritmul de adunare peste întregi și $*_{Int}$ reprezintă algoritmul de înmulțire peste întregi.

Reguli de calcul pentru $\llbracket \cdot \rrbracket (-)$:

$$\begin{array}{c}
\frac{}{\llbracket \mathbf{x} \rrbracket (\dots \mathbf{x} \mapsto \mathbf{v} \dots) = \mathbf{v}} \quad \frac{}{\llbracket \mathbf{v} \rrbracket (-) = \mathbf{v}} \\
\frac{\llbracket E_1 \rrbracket (\sigma) = v_1 \quad \llbracket E_2 \rrbracket (\sigma) = v_2}{\llbracket E_1 + E_2 \rrbracket (\sigma) = v_1 +_{Int} v_2} \dots
\end{array}$$

Calculul timpului pentru evaluare

$time_d(\llbracket \mathbf{a} + \mathbf{b} * 2 \rrbracket(\sigma)) = time_d(\llbracket \mathbf{a} \rrbracket(\sigma)) + time_d(\llbracket \mathbf{b} \rrbracket(\sigma)) + time_d(6 *_{Int} 2) + time_d(3 +_{Int} 122)$, $d \in \{\text{unif}, \text{log}\}$. $time_{\text{log}}(\llbracket \mathbf{a} \rrbracket(\sigma)) = \log 3$, $time_{\text{log}}(\llbracket \mathbf{b} \rrbracket(\sigma)) = \log 6$ $time_{\text{unif}}(\llbracket \mathbf{a} \rrbracket(\sigma)) = 1$, $time_{\text{unif}}(\llbracket \mathbf{b} \rrbracket(\sigma)) = 1$

Regulile de mai sus ne dau si o metoda de calcul a timpului necesar evaluării:

$time(\llbracket \mathbf{x} \rrbracket(\dots \mathbf{x} \mapsto \mathbf{v} \dots)) = \log \mathbf{v}$

$time(\llbracket \mathbf{x} \rrbracket(\dots \mathbf{x} \mapsto \mathbf{v} \dots)) = 1$ (*uniform*)

$time(\llbracket E_1 + E_2 \rrbracket(\sigma)) = time(\llbracket E_1 \rrbracket(\sigma)) + time(\llbracket E_2 \rrbracket(\sigma)) + time(\llbracket E_1 \rrbracket(\sigma) +_{Int} \llbracket E_2 \rrbracket(\sigma)) \dots$

Configurații

O configurație este o pereche $\langle \text{secvență-de-program}, \text{stare} \rangle$

Exemple:

$\langle \mathbf{x} = \mathbf{x} + 1; \mathbf{y} = \mathbf{y} + 2 * \mathbf{x};, \mathbf{x} \mapsto 7 \mathbf{y} \mapsto 12 \rangle [1\text{ex}]$

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

Pași de execuție

Un pas de execuție este definit ca o tranziție între configurații:

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

dacă și numai dacă

executând prima instrucțiune secvența din S în starea σ obținem secvența S' , ce urmează a fi executată în continuare, și o nouă stare σ'

Pașii de execuție sunt descriși 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 execuție, vom preciza costul dat de aplicarea unei reguli.

Instrucțiuni: semantica

atribuirea: $\mathbf{x} = E$;

- *informal*: se evaluează E și rezultatul este atribuit ca noua valoare a variabilei \mathbf{x}
- *formal*: $\langle \mathbf{x} = E; S, \sigma \rangle \Rightarrow \langle S, \sigma' \rangle$ unde σ este de forma $\dots \mathbf{x} \mapsto v \dots$ și σ' de forma $\dots \mathbf{x} \mapsto \llbracket E \rrbracket(\sigma) \dots$ (în rest la fel ca σ).

Costul timp: $time_{\text{unif}}(\langle \mathbf{x} = E; S, \sigma \rangle \Rightarrow \langle S, \sigma' \rangle) = time_{\text{unif}}(\llbracket E \rrbracket(\sigma)) + 1$,
 $time_{\text{log}}(\langle \mathbf{x} = E; S, \sigma \rangle \Rightarrow \langle S, \sigma' \rangle) = time_{\text{log}}(\llbracket E \rrbracket(\sigma) + \log \llbracket E \rrbracket(\sigma))$.

Instrucțiuni: semantica

if: $\text{if } (E) \text{ then } S \text{ else } S'$

- *informal*: se evaluează e ; dacă rezultatul obținut este *true*, atunci se execută S , altfel se execută S'

- *formal*:

$\langle \text{if } (E) \text{ then } S \text{ else } S' \ S'', \sigma \rangle \Rightarrow \langle S \ S'', \sigma \rangle$ dacă $\llbracket E \rrbracket(\sigma) = \text{true}[1\text{ex}]$ $\langle \text{if } (E) \text{ then } S \text{ else } S' \ S'', \sigma \rangle \Rightarrow \langle S' \ S'', \sigma \rangle$ dacă $\llbracket E \rrbracket(\sigma) = \text{false}$

Costul timp:

$$time_d(\langle \text{if } (E) \text{ then } S' \text{ else } S'' \ S, \sigma \rangle \Rightarrow \langle -, \sigma \rangle) = time_d(\llbracket E \rrbracket(\sigma)) \quad d \in \{\text{unif}, \log\}.$$

Instrucțiuni: semantica

while: **while** (E) S

- *informal*: se evaluează e ; dacă rezultatul obținut este *true*, atunci se execută S , după care se evaluează din nou e și ...; altfel execuția instrucțiunii se termină

- *formal*: se exprimă cu ajutorul lui **if**:

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

Costul timp:

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

Apelul de funcție

Operație mai complexă ce include mai multe activități:

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

Presupunem funcția $f(a, b) \ \{ S_f \}$. Evaluarea apelului $f(e_1, e_2)$ presupune următorii pași: [lex]

$$\begin{aligned} \langle f(e_1, e_2) \ S, \sigma, \text{Stack} \rangle &\Rightarrow [\text{lex}] \\ \langle S_f, \sigma \cup \{a \mapsto \llbracket e_1 \rrbracket(\sigma) \ b \mapsto \llbracket e_2 \rrbracket(\sigma)\}, (S, \sigma) \ \text{Stack} \rangle &\Rightarrow^* [\text{lex}] \\ \langle v, \sigma', (S, \sigma) \ \text{Stack} \rangle &\Rightarrow [\text{lex}] \\ \langle v \ S, \text{updateGlobals}(\sigma, \sigma'), \text{Stack} \rangle &\end{aligned}$$

Presupunere: costul unui apel este suma dintre costul evaluării argumentelor actuale și costul execuției corpului funcției.

Calcul (execuție)

Un calcul (o execuție) este o secvență de pași: $\tau = \langle S_1, \sigma_1 \rangle \Rightarrow \langle S_2, \sigma_2 \rangle \Rightarrow \langle S_3, \sigma_3 \rangle \Rightarrow \dots$

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

Calcul: exemplu

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

Am utilizat: $\llbracket x > 3 \rrbracket(x \mapsto 7 \ y \mapsto 12) = \text{true}$
 $\llbracket x + y \rrbracket(x \mapsto 7 \ y \mapsto 12) = 19$
 $\llbracket 4 \rrbracket(x \mapsto 19 \ y \mapsto 12) = 4$

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

Calcul: exemplu

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

Am utilizat: $\llbracket i > 5 \rrbracket(i \mapsto 6 \ x \mapsto 12) = \text{true}$
 $\llbracket i-- \rrbracket(i \mapsto 6 \ x \mapsto 12) = 0$
 $\llbracket i > 5 \rrbracket(i \mapsto 5 \ x \mapsto 12) = \text{false}$

Costul timp: cost uniform: 4 (= numărul de pași, 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 definiții de limbaje de programare. Definițiile K sunt executabile. Limbajul Alk este definit în K, așa că algoritmi descriși în Alk pot fi testați și analizați.

Definiția K a lui Alk se găsește la adresa <https://github.com/kframework/alk-semantics> și poate fi compilată cu K versiunea 3.6: <https://github.com/kframework/k/releases>. O copie actualizată va fi accesibilă de pe pagina cursului.

Compilarea definiției limbajului Alk: *kompile alk*

Proiect (posibil pentru licență): dezvoltarea unei interfețe specifice pentru Alk.

Execuția algoritmului DFS recursiv

Starea inițială este dată ca valoare a opțiunii cINIT:

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

Obținerea configurației finale

Configurația finală obținută prin comanda precedentă este:

```
<k>
.K
</k>
<state>
  D |-> { n -> 3 a -> ([ (< 1, 2 >), (< 2, 0 >), (< 0, .ValueList >) ]) }
  S |-> [ 1, 1, 1 ]
  dfsRec |-> lambda ( i ) . ({ if ( S [ i ] == 0 ) { ((S [ i ] = 1) ;) ((p
    = (D . a) [ i ]) ;) (while ( p . size ( ) > 0 ) { ((j = p . topFront
      ( )) ;) (p . popFront ( ) ;) (dfsRec ( j ) ;) } } })
  i0 |-> 1
  i |-> 3
</state>
```

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

Demo

Execuția algoritmilor de mai sus.