Acest document este similar secțiunii 2.7 din Cristian Giumale, "Introducere în analiza Algoritmilor", conținutul fiind extins cu:

1. Specificarea formală a tipului `T Ring`
2. Verificarea corectitudinii specificației `T Ring`
3. Verificarea implementării tipului `T Ring` ca o pereche de liste `T List`
4. Rescrierea Haskell a specificației `T Ring` şi implementarea
5. $\texttt{T Ring} = \texttt{Abs}_{D}\texttt{(T List, T List)}$
6. Dezvoltarea unui algoritm corect pentru sortarea unei liste de chei folosind un inel.

2.7 Analiză amortizată

Tehnica de analiză a complexității, aşa cum a fost discutată în secțiunile precedente, nu ține seama de posibila comportare neuniformă a algoritmului studiat. Analiza este realizată punctual, pentru un singur pachet de date, şi, de regulă, pentru cazul cel mai defavorabil. Este însă posibil ca un algoritm să fie ineficient în cazurile defavorabile, dar performant în celelalte cazuri. Dacă situațiile defavorabile apar relativ rar, atunci pe ansamblu, pentru o mulțime de pachete de date, algoritmul poate avea o comportare acceptabilă, iar costul secvenței de operații efectuate de algoritm poate rezulta scăzut. Astfel, devine interesant studiul costurilor secvențelor posibile de operații ce pot fi efectuate de algoritm, cu condiția ca secvențele să acopere cazurile defavorabile. Analiza complexității amortizate (sau pe scurt analiza amortizată) a unui algoritm are exact acest scop: determinarea marginii superioare a costului unei secvențe oarecare de operații şi, totodată, calculul costului mediu al operațiilor algoritmului.

Costul mediu **$cost_a$(op)** al unei operații **op**, calculat pentru o secvență oarecare **S** de operații ce acoperă cazurile cele mai defavorabile, se numeşte *cost amortizat* al operației, iar suma $\sum_{op \in S} cost_a(op)$ este o margine superioară a costului secvenței **S**.

2.7.1 Inel unidirecțional

Ca bază pentru discutarea metodelor de analiză amortizată, se alege un inel unidirecțional cu elemente de tip `T`. Un inel nevid este caracterizat printr-un vârf ce poate fi deplasat prin inel element cu element astfel încât doar elementul de la vârful inelului este accesibil. Evident, după parcurgerea tuturor elementelor din inel, vârful ajunge în poziția inițială. Considerăm că operațiile inelului sunt:

- `void:`→ `T Ring` construieşte inelul vid.
- `ins:T` × `T Ring` → `T Ring`. `ins(e,R)` inserează elementul `e` la vârful inelului `R`.
- `del:T Ring\{void}` → `T Ring`. `del(R)` elimină elementul de la vârful inelului nevid `R`. Vârful este poziționat pe următorul element, dacă acesta există.
- `top:T Ring\{void}` → `T`. `top(R)` este elementul de la vârful inelului nevid `R`.
- `move:T Ring\{void}` → `T Ring`. Prin `move(R)` vârful inelului nevid `R` este deplasat spre dreapta, peste elementul curent.
- `empty:T Ring` → `bool = {0,1}`. `empty(R)` testează dacă inelul `R` este vid.

Descrierea informală de mai sus corespunde următoarei specificații algebrice a tipului `T Ring`.

```
T Ring ::= void | ins(T, T Ring)
-------------------------------
empty: T Ring → bool
top: T Ring \{void} → T
del: T Ring\{void} → T Ring
move: T Ring\{void} → T Ring
-------------------------------
(1) empty(void) = 1
(2) empty(ins(e,X)) = 0
(3) top(ins(e,X)) = e
(4) del(ins(e,X)) = X
(5) move(ins(e,void)) = ins(e,void)
(6) move(ins(e,ins(e',X))) = ins(e',move(ins(e,X)))
```

Cum verificăm dacă specificația corespunde într-adevăr unui inel unidirecțional? Cu alte cuvinte, cum putem proba corectitudinea specificației? O posibilitate constă în alegerea unei proprietăți reprezentative inelului şi în a o verifica folosind specificația. De exemplu, $\forall i \in \mathbb{N}, X \in$ `T Ring | ¬empty(X) • top(move`^{i}`(X)) = element(i,X)`, unde `move`^{i}`(X)` reprezintă aplicarea de `i` ori a operației `move` pornind cu inelul `X`, `element(i,X)` reîntoarce elementul `i modulo size(X)` din `X`, iar `size(X)` reîntoarce numărul elementelor din `X`. Un asemenea exercițiu este efectuat în secțiunea 2.4.5.

Inelul poate fi implementat ca o listă circulară, iar prelucrarea este banală dacă limbajul de programare folosit lucrează direct cu pointerii din celulele listei. În acest caz toate operațiile sunt în $\Theta(1)$. Să considerăm însă că limbajul de programare folosit nu lucrează în mod direct cu pointeri ci dispune de tipul `T List` (listă unidirecțională cu elemente de tip `T`), cu operatorii şi specificația formală de mai jos, unde:

- `nil` este lista vidă.
- `cons(e,L)` construieşte din lista `L` o listă care are ca prim element `e`.
- `head(L)` reîntoarce elementul de la vârful listei nevide `L`.
- `tail(L)` reîntoarce lista rezultată prin decuparea elementului de la vârful lui `L`.
- `null(L)` testează dacă lista `L` este vidă.

```
T List ::= nil | cons(T, T List)
-----------------------------------
head: T List\{nil} → T
tail: T List\{nil} → T List
null: T List → bool
-----------------------------------
(1L) head(cons(e,L)) = e
(2L) tail(cons(e,L)) = L
(3L) null(nil) = 1
(4L) null(cons(e,L)) = 0
```

O implementare posibilă a tipului `T Ring` folosind tipul `T List` consideră un inel nevid ca abstractizarea $\mathtt{Abs_p}\langle \mathtt{L,R}\rangle$ a unei perechi $\langle \mathtt{L,R}\rangle$ de liste unidirecționale `L` şi `R`. Lista `R` conține elementele din inel situate între vârful curent şi "ultimul" element al inelului, iar `L` conține elementele, aranjate în ordine inversă, aflate între poziția inițială şi poziția curentă a vârfului inelului. Inelul vid este abstractizat prin $\mathtt{Abs_p}\langle \mathtt{nil,nil}\rangle$. Reprezentarea este ilustrată în figura 2.8.

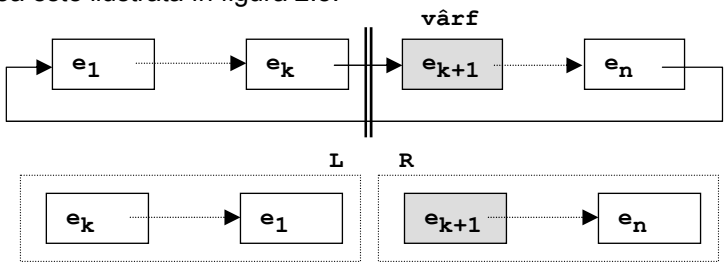

Figura 2.8 Un inel unidirecțional reprezentat prin două liste unidirecționale

Fie inelul $\mathtt{x} = \mathtt{Abs_p}\langle \mathtt{L,R}\rangle$. Convenim ca lista `R` să fie vidă doar dacă inelul `x` este vid. O astfel de reprezentare a inelului o vom numi normalizată şi poate fi obținută prin operația:

$$\mathtt{(norm)\ normalize}\langle \mathtt{L,R}\rangle = \mathtt{if\ null(R)\ then}\ \langle \mathtt{nil,reverse(L)}\rangle\ \mathtt{else}\ \langle \mathtt{L,R}\rangle$$

unde operația `reverse(L)` construieşte inversa listei `L`. Dacă inversarea unei liste este fi definită conform axiomelor

```
reverse(L) = rev(L,nil), CU rev(nil,R) = R
                            rev(cons(e,L),R) = rev(L,cons(e,R)),
```

atunci complexitatea operației `reverse(L)` este $\Theta(n)$, `n` fiind numărul de elemente din lista `L`. Aşadar, în cazul cel mai defavorabil, complexitatea operației $normalize\langle L,R\rangle$ este proporțională cu numărul de elemente din inelul $Abs_p\langle L,R\rangle$.

Operațiile inelului presupun un inel cu reprezentare normalizată şi conduc la un inel cu reprezentare normalizată, fiind implementate conform ecuațiilor următoare, numite axiome de implementare:

$$
\begin{aligned}
&(1i)\ void = Abs_p\langle nil,nil\rangle\\
&(2i)\ ins(e,Abs_p\langle L,R\rangle) = Abs_p\langle L,cons(e,R)\rangle\\
&(3i)\ del(Abs_p\langle L,R\rangle) = Abs_p(normalize\langle L,tail(R)\rangle)\\
&(4i)\ move(Abs_p\langle L,R\rangle) = Abs_p(normalize\langle cons(head(R),L),tail(R)\rangle)\\
&(5i)\ top(Abs_p\langle L,R\rangle) = head(R)\\
&(6i)\ empty(Abs_p\langle L,R\rangle) = null(R)
\end{aligned}
$$

Este interesant de observat că funcția Abs_p este definită peste clase de echivalență ale perechilor $\langle L,R\rangle$. Într-adevăr, în cursul prelucrărilor, se pot obține implementări diferite – dar echivalente – ale aceluiaşi inel. Spunem că reprezentarea $\langle L,R\rangle$ este echivalentă cu reprezentarea $\langle L',R'\rangle$ a unui inel, cu alte cuvinte spunem că $Abs_p\langle L,R\rangle$ şi $Abs_p\langle L',R'\rangle$ reprezintă acelaşi inel şi scriem $Abs_p\langle L,R\rangle = Abs_p\langle L',R'\rangle$, dacă `R @ (reverse L) = R' @ (reverse L')`, unde `@` reprezintă concatenarea listelor. Dacă `Z` şi `W` sunt valori de tipul `T List`, iar `e` este un element de tip `T`, concatenarea poate fi implementată conform axiomelor:

```
nil @ Z = Z
cons(e,W) @ Z = cons(e, W @ Z)
```

Folosind relația de echivalență

$$Abs_p\langle L,R\rangle = Abs_p\langle L',R'\rangle \Leftrightarrow R\ @\ (reverse\ L) = R'\ @\ (reverse\ L'),$$

se poate verifica că implementarea conformă ecuațiilor de mai sus respectă axiomele din specificarea algebrică a tipului `T Ring`, deci că implementarea este corectă. De exemplu, fie $X = Abs_p\langle L,R\rangle$ şi axioma

```
move(ins(e,ins(e',X))) = ins(e',move(ins(e,X)))
```

Prin rescrierea celor doi termeni ai axiomei, conform axiomelor tipurilor `T Ring`, `T List` şi ecuațiilor de implementare, se ajunge la acelaşi inel.

$$
\begin{aligned}
&move(ins(e,ins(e',X)))\\
&\quad = move(ins(e,ins(e',Abs_p\langle L,R\rangle)))\\
&\quad \text{—— 2i} \rightarrow move(ins(e,Abs_p\langle L,cons(e',R)\rangle))\\
&\quad \text{—— 2i} \rightarrow move(Abs_p\langle L,cons(e,cons(e',R))\rangle)\\
&\quad \text{—— 4i,1L,2L} \rightarrow Abs_p(normalize\langle cons(e,L),cons(e',R)\rangle)\\
&\quad \text{—— norm} \rightarrow Abs_p\langle cons(e,L),cons(e',R)\rangle\\
&\quad = X'
\end{aligned}
$$

```
ins(e',move(ins(e,X))
      = ins(e',move(ins(e,Abs_p⟨L,R⟩)))
      ⟶ 2i → ins(e',move(Abs_p⟨L,cons(e,R)⟩))
      ⟶ 4i,1L,2L → ins(e',Abs_p(normalize⟨cons(e,L),R⟩))
```

Caz 1. `R = nil`, `L = nil`

```
      ⟶ norm → ins(e',Abs_p⟨nil,cons(e,nil)⟩)
      ⟶ 2i → Abs_p⟨nil,cons(e',cons(e,nil))⟩
      = X"
```

Dar, în acest caz, `X' = Abs_p⟨cons(e,nil),cons(e',nil)⟩`, iar `cons(e',nil) @ reverse(cons(e,nil)) = cons(e',cons(e,nil))`. Deci `X' = X"`.

Caz 2. `R ≠ nil`

```
      ⟶ norm → ins(e',Abs_p⟨cons(e,L),R⟩)
      ⟶ 2i → Abs_p⟨cons(e,L),cons(e',R)⟩
      = X'
```

2.7.2 Analiză de ansamblu

Costul operațiilor **del** şi **move** este $O(n)$ în cazul cel mai defavorabil, unde n este lungimea inelului, dar cazul cel mai defavorabil nu apare mereu. Dificultatea stabilirii exacte a complexității operațiilor rezidă în interdependența dintre frecvența cazurilor defavorabile, când normalizarea are loc efectiv, şi secvența de operații aplicate inelului. Prin urmare o asemenea complexitate exactă, absolută, este imposibil de calculat. Ce se poate calcula este costul mediu al unei secvențe oarecare de operații. Luând ca operație critică deplasarea unui element al inelului între listele **L** şi **R**, un raționament posibil este următorul.

Aparent, pornind de la un inel vid, după o secvență **s** de m operații oarecare fiecare element din inel poate fi deplasat între **L** şi **R** de cel mult $O(m)$ ori. Deplasarea **L->R** are loc în timpul unei normalizări, iar deplasarea **R->L** are loc în prima fază a unei operații **move**, înaintea eventualei normalizări. Inelul poate conține cel mult m elemente, deci costul celor m operații este $cost(S) \le m\,O(m) = O(m^2)$. Prin urmare, costul amortizat al unei operații din cele m este $O(m^2)/m = O(m)$. Totuşi, acest rezultat reprezintă o margine superioară mult mai largă decât cea reală. Normalizările efective, deci transferurile **L->R** nu au loc pentru fiecare operație din cele m. De asemenea, dacă inelul conține m elemente atunci nici o operație nu poate fi **move** sau **del**.

Pentru o estimare mai exactă a costului amortizat al operațiilor din secvența **s**, să observăm că numărul de deplasări **L->R** de elemente nu poate fi mai mare ca numărul operațiilor **move** din **s**. Într-adevăr, un element trebuie să fie în lista **L** pentru a putea fi transferat în **R** şi numai operațiile **move** încarcă lista **L**. De asemenea, numărul deplasărilor **R->L** este egal cu numărul operațiilor **move** din **s**. Prin urmare, numărul total de deplasări de elemente între listele **L** şi **R** ale inelului nu poate depăşi dublul

numărului operațiilor din **S**. Costul cumulat al acestor deplasări (ca efect al operațiilor **move** şi **del** din **S**) este **O(2m)**. La costul deplasărilor de elemente se adaugă costul inserărilor elementelor în **R**, ca efect al operațiilor **ins**, costul eliminărilor elementelor din **R**, ca efect al operațiilor **del** şi costul operațiilor **top** şi **empty**. Costul tuturor acestor operații este **O(m)**. Rezultă **cost(S)= O(m)+O(2m)= O(m)**, iar costul amortizat al unei operații a inelului este **O(m)/m = O(1)**.

Metoda utilizată mai sus pentru a estima costul mediu al unei operații a inelului unidirecțional poartă numele de analiză de ansamblu (în engleză metoda este întâlnită sub denumirea *aggregate method*). Se urmăreşte contorizarea execuțiilor tuturor suboperațiilor ce formează operațiile unei structuri de date în raport cu lungimea unei secvențe oarecare de astfel de operații. Metoda calculează un singur *cost amortizat* (cost mediu al operațiilor unei secvențe oarecare de operații care acoperă cazurile cele mai defavorabile) pentru toate operațiile unui algoritm, chiar dacă operațiile au costuri amortizate diferite.

2.7.3 Metoda creditelor

O analiză amortizată mai fină impune diferențierea costurilor diverselor operații efectuate de algoritm. O astfel de analiză poate folosi metoda creditelor (în engleză metoda este întâlnită sub denumirea *accounting method*).

Presupunem că algoritmul execută o secvență de operații **s** asupra unei structuri de date **D**. Fiecărei operații **op** din **s** i se asociază un cost amortizat $cost_a(op)$, costul real al operației fiind notat **cost(op(D))**. Valoarea $cost_a(op)$ este costul mediu al **op**, cost care acoperă cazurile cele mai defavorabile, în timp ce **cost(op(D))** poate varia în funcție de istoria prelucrării şi de elementele prelucrate ale structurii **D**. Atunci când $cost_a(op) > cost(op(D))$ diferența $cost_a(op) - cost(op(D))$ reprezintă un credit care măreşte creditul **credit(e)** deja asociat elementelor $e \in D$ implicate în operația **op**. Creditul este utilizat pentru a compensa costul real al acelor operații **op** din **s** care satisfac $cost(op(D)) > cost_a(op)$.

Să impunem restricția $credit(e) \geq 0$, pentru orice moment de timp în cursul operațiilor din **s** şi pentru orice element din **D**. Să notăm cu $credit_f(e)$ creditul folosit pentru a compensa costul amortizat al operațiilor $op \in s$ care satisfac $cost_a(op) < cost(op(D))$, şi cu $credit_c(e)$ creditul cumulat al unui element **e** din **D**, obținut în urma operațiilor $op \in s$ cu cost amortizat $cost_a(op) > cost(op(D))$. Putem scrie:

$$\sum_{e \in D} credit_c(e) - \sum_{e \in D} credit_f(e) = \sum_{op \in S} cost_a(op) - \sum_{op \in S} cost(op(D)) \geq 0$$

Astfel, se poate garanta că pentru orice secvență **s** de operații există inegalitatea $cost_a(S) = \sum_{op \in S} cost_a(op) \geq \sum_{op \in S} cost(op(D)) = cost(S)$, adică suma

costurilor amortizate ale operațiilor din secvență reprezintă o limită superioară a costului real al secvenței.

Pentru inelul unidirecțional costurile reale ale operațiilor **ins**, **top** şi **empty** sunt **O(1)** şi, pentru simplitate, sunt considerate **1**. De asemenea, considerăm **1** costul unei operații de deplasare a unui element între listele **L** şi **R** ale inelului. Costul real pentru **move** depinde de starea inelului şi variază între **1** şi **1+n**, unde **n** este numărul de elemente din inel. Pentru **del** costul real este între **1** şi **n**. Alocăm costurile amortizate:

$$cost_a(void) = cost_a(ins) = cost_a(del) = cost_a(top) = cost_a(empty) = 1$$
$$cost_a(move) = 2$$

Să asociem fiecărui element **e** din inel un credit **credit(e)** astfel încât creditul inițial să fie **0**. Doar operatorii **del** şi **move** pot să modifice creditul elementelor unui inel. De exemplu, după operația **move(X)** fără normalizare creditul elementului **top(X)** creşte cu diferența dintre $cost_a(move) - cost(move(X))$, adică cu **1**. Distribuția creditelor elementelor unui inel este precizată de invariantul:

$$I(Abs_p\langle L,R\rangle) =_{def} (\forall e \in L \bullet credit(e)=1) \wedge (\forall e \in R \bullet credit(e)=0)$$

Invariantul arată că pentru orice element din inel creditul rămâne mereu pozitiv. În ipoteza în care operațiile ce nu modifică inelul nu modifică nici creditele elementelor, iar costurile amortizate asociate operațiilor care modifică inelul respectă invariantul, adică

$$I(void) \wedge (\forall X \in T\ Ring \bullet I(X) \Rightarrow (\forall op \in \{ins, del, move\} \bullet I(op(X)))),$$

putem admite că, pentru orice secvență de operații şi pentru orice inel, costul mediu al unei operații **op** este $cost_a(op)$. Pentru constructorii **void**, **ins**, **move** şi **del** demonstrația este prin inducție structurală.

Propoziția 2.6 Costurile amortizate ale constructorilor inelului $Abs_p\langle L,R\rangle$ mențin invariantul $I(Abs_p\langle L,R\rangle)$.

Caz de bază. $X = void = Abs_p\langle nil,nil\rangle$, iar invariantul **I(X)** este satisfăcut banal. În plus, $cost_a(void) = cost(void)$ şi deci nu există credit ce trebuie distribuit.

Pas de inducție. Să considerăm că pentru inelul $X = Abs_p\langle L,R\rangle$, $R \neq nil$, cu **n** elemente, invariantul **I(X)** este satisfăcut (ipoteza inductivă). Să arătăm că invariantul este satisfăcut după orice operație **ins**, **del** sau **move** aplicată lui **X**.

• Pentru operația **ins(e,X)** inelul devine $Abs_p\langle L, cons(e,R)\rangle$. Costul amortizat $cost_a(ins)$ este **1** exact cât $cost(ins(e,X))$ astfel încât $credit(e)=0$ şi, conform ipotezei inductive, invariantul $I(Abs_p\langle L, cons(e,R)\rangle)$ este satisfăcut.

• Pentru operația $move(X) = Abs_p\langle L',R'\rangle$ fie **e** elementul de la vârful lui **X**, anume $e = top(X) = head(R)$.

Caz 1: `tail(R)≠nil`. Operația `move` are o singură fază: deplasarea lui `e` din `R` în `L`, astfel încât `L'= cons(e,L)` şi `R'= tail(R)`. Costul real al operației `move(X)` este `1`, iar diferența $cost_a$`(move)-cost(move(X)) = 2-1 = 1` este adăugată creditului elementului `e`. Conform ipotezei inductive, înaintea operației `move` avem `credit(e)=0`, astfel încât `credit(e)=1` după `move(X)`. Invariantul $I(Abs_p\langle L',R'\rangle)$ este satisfăcut.

Caz 2: `tail(R)=nil`. Operația `move` are două faze: transferul lui `e` din `R` în `L` şi apoi normalizarea efectivă a inelului, astfel încât `L'= nil` şi `R'= reverse(cons(e,L))`.

a) Elementul `e` este mutat din `R` în `L`. Ca urmare a ipotezei inductive, creditul lui `e` devine `credit(e)=1`, iar elementele din `L` au creditul `1`. Invariantul $I(Abs_p\langle cons(e,L),nil\rangle)$ este satisfăcut după această primă fază.

b) Inelul este normalizat. Toate elementele din `cons(e,L)` sunt deplasate în `R'`. Costul deplasării unui element `e` este `1` şi este suportat din creditul `credit(e)=1`, astfel încât, odată ajuns în `R'`, elementul `e` are `credit(e)=0`. Invariantul $I(Abs_p\langle nil,reverse(cons(e,L))\rangle = I(move(X))$ este satisfăcut.

În cazul (2) există `n-1` elemente în lista `L` şi un singur element în lista `R`, iar costul real al operației `move(X)` este `1+n`,. Deşi $cost_a$`(move)=2`, diferența `n-1` este suportată din creditele elementelor aflate în `L` înaintea operației `move`.

• Pentru `del(X)` $= Abs_p\langle L',R'\rangle$, notăm `e = top(X) = head(R)`. Ca şi pentru `move(X)` există două cazuri.

Caz 1: `tail(R)≠nil`. Operația `del` are o singură fază: eliminarea lui `e` din `R`. Rezultă `L'= L` şi `R'= tail(R)`. Costul real al operației `del(X)` este `1` exact cât costul amortizat, deci nu au loc modificări de credite. Conform ipotezei inductive, invariantul $I(Abs_p\langle L,tail(R)\rangle)$ este satisfăcut.

Caz 2: `tail(R)=nil`. Operația `del` are două faze: eliminarea elementului `e` din `R` şi normalizarea efectivă a inelului. Rezultă `L'=nil` şi `R'= reverse(L)`.

a) Elementul `e` este eliminat din `R`. Operația costă `1` exact cât costul amortizat $cost_a$`(del)`. Nu au loc modificări de credite şi, conform ipotezei inductive, invariantul $I(Abs_p\langle L,nil\rangle)$ este satisfăcut.

b) Inelul este normalizat. Elementele din `L` sunt deplasate în `R'`. Costul deplasării unui element `e` este `1` şi este suportat din creditul `credit(e)=1`, astfel încât, odată ajuns în `R'`, elementul `e` are `credit(e)=0`. Invariantul $I(Abs_p\langle nil,reverse(L)\rangle) = I(del(X))$ este satisfăcut.

În cazul (2) costul real al operației `del(X)` este `n`. Deşi $cost_a$`(del)=1`, diferența `n-1` este suportată din creditele celor `n-1` elemente aflate în `L` înaintea operației. ■

Deoarece suma creditelor elementelor din inel este întotdeauna pozitivă, rezultă că pentru o secvență oarecare `s` de `m` operații ce pleacă cu un inel vid `X` avem:

$$cost(S) = \sum_{op \in S} cost(op(D)) \leq \sum_{op \in S} cost_a(op) \leq 2m = O(m)$$

Costul mediu al unei operații `op` a inelului, cost ce acoperă cazurile cele mai defavorabile, este $cost_a(op) = O(1)$, deşi costul real poate varia neuniform aşa cum se ilustrează în figura 2.9. Totodată, analiza complexității amortizate este independentă în raport cu starea inițială $x_0 = Abs_p\langle L_0, R_0\rangle$ a inelului, cu condiția ca invariantul $I(Abs_p\langle L_0, R_0\rangle)$ să fie satisfăcut.

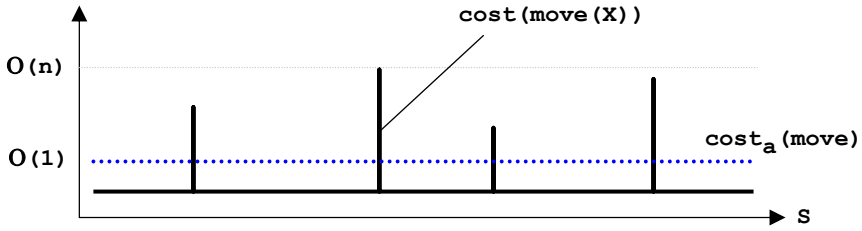

Figura 2.9 Costul real şi costul amortizat pentru operația `move` (`n` este lungimea maximă a inelului în cursul operațiilor `s`)

2.7.4 Metoda potențialului

Metoda potențialului derivă din cea a creditelor prin centralizarea acestora. În loc de a asocia credite elementelor structurii de date prelucrate de algoritm, putem asocia un singur credit întregii structuri de date. În acest caz creditul joacă rolul unei energii potențiale care poate folosi pentru compensarea operațiilor mai costisitoare. Metoda rezultată este denumită metoda potențialului.

Structurii de date `D` i se asociază o funcție $\Phi: D \rightarrow R_+$, care măsoară energia potențială a întregii structuri la un moment dat. Fie $s = op_1, op_2, \ldots, op_m$ o secvență de `m` operații executate de un algoritm asupra structurii `D`. Fie D_0 structura inițială, înaintea aplicării operației op_1, şi D_i structura rezultată după aplicarea primelor `i` operații asupra lui D_0. De asemenea, notăm cu Φ_i energia potențială a lui D_i.

$$D_i =_{def} op_i(op_{i-1}(\ldots op_1(D_0)\ldots)) = op_i(D_{i-1}), \; i=1,m$$
$$\Phi_i =_{def} \Phi(D_i), \; i=0,m$$

Fie $cost(op_i)$ costul real al operației $op_i(D_{i-1})$, iar $cost_a(op_i)$ costul amortizat al operației op_i relativ la Φ, cost definit prin:

$$cost_a(op_i) =_{def} cost(op_i) + \Phi_i - \Phi_{i-1}, \; i=1,m$$

Costul amortizat al operației $op_i(D_{i-1})$ adaugă costului real variația de potențial a structurii de date `D`, variație indusă de operație. Însumând, obținem:

$$\sum_{i=1}^{m} cost_a(op_i) = \sum_{i=1}^{m} (cost(op_i) + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{m} cost(op_i) + \Phi_m - \Phi_0$$

Dacă $\Phi_m \geq \Phi_0$ atunci $\sum_{i=1}^{m} cost_a(op_i) \geq \sum_{i=1}^{m} cost(op_i)$ şi, deoarece costul secvenței s este suma costurilor operațiilor din s, rezultă

$$cost(S) \leq \sum_{i=1}^{m} cost_a(op_i) = cost_a(S).$$

Dorim păstrarea inegalității $cost(S) \leq cost_a(S)$ pentru orice secvență s, implicit, pentru orice m şi impunem inegalitatea $\Phi_i \geq \Phi_0$ pentru orice i. Cu alte cuvinte, energia potențială a structurii D la un moment diferit celui inițial trebuie să fie mai mare decât energia inițială. Pentru cazul particular $\Phi_0=0$, este necesar ca $\Phi_i \geq 0$. Această ultimă inegalitate este naturală, din moment ce energia nu poate fi consumată în lipsă.

Ca exemplu, să reconsiderăm inelul unidirecțional reprezentat printr-o pereche de liste, ca în figura 2.8. Definim funcția Φ asociată inelului $X = Abs_p\langle L,R\rangle$ prin $\Phi(X) = \#L$, unde #L desemnează numărul de elemente din lista L. Funcția de potențial respectă restricțiile:

$\Phi(X) = 0$ pentru un inel vid $X = Abs_p\langle nil,nil\rangle$
$\Phi(X) \geq 0$ pentru un inel oarecare $X = Abs_p\langle L,R\rangle$

Putem calcula acum costurile amortizate ale operațiilor ce modifică inelul X. Suma acestor costuri pentru o secvență de operații s, aplicată pe un inel inițial vid (sau pe un inel cu #L=0), constituie o margine superioară a costului real al secvenței. Notăm $X_i = Abs_p\langle L_i, R_i\rangle$ inelul X obținut după a i-a operație din secvența s.

• Presupunând că ins este a i-a operație din secvența de operații s executate asupra inelului X, anume $X_i = ins(e,X_{i-1})$, avem: $cost(ins(e,X_{i-1})) = 1$, $X_i = Abs_p\langle L_{i-1}, cons(e,R_{i-1})\rangle$, şi $\Phi_i = \#L_i = \#L_{i-1} = \Phi_{i-1}$. Rezultă,

$$cost_a(ins) = cost(ins(e,X_{i-1})) + \Phi_i - \Phi_{i-1} = 1$$

• Presupunând că move este a i-a operație din secvența de operații s, anume $X_i = move(X_{i-1})$, există două cazuri posibile.

Caz $\#R_{i-1} > 1$. Operația $move(X_{i-1})$ constă în transferul primului element din R_{i-1} astfel încât $L_i = cons(top(R_{i-1}), L_{i-1})$ şi $R_i = tail(R_{i-1})$.

$$cost(move(X_{i-1})) = 1$$
$$\Phi_i = \#L_i = \#L_{i-1} + 1$$
$$\Phi_{i-1} = \#L_{i-1}$$
$$cost_a(move) = cost(move(X_{i-1})) + \Phi_i - \Phi_{i-1} = 2$$

Caz $\#R_{i-1} = 1$. După transferul unicului element $e=top(R_{i-1})$ din R_{i-1} are loc normalizarea efectivă a inelului $X' = Abs_p\langle cons(e,L_{i-1}),nil\rangle$. Rezultă, $Abs_p\langle L_i,R_i\rangle = Abs_p\langle nil,reverse(cons(e,L_{i-1}))\rangle$. Normalizarea inelului X' costă $\#L_{i-1}+1$.

$$cost(move(X_{i-1})) = 1 + cost\_normalizare(X') = 1 + \#L_{i-1} + 1$$
$$\Phi_i = \#L_i = \#nil = 0$$
$$\Phi_{i-1} = \#L_{i-1}$$
$$cost_a(move) = cost(move(X_{i-1})) + \Phi_i - \Phi_{i-1} = 2$$

• Presupunând că del este a i-a operație din secvența de operații s, anume $X_i = del(X_{i-1})$, există două cazuri posibile.

Caz $\#R_{i-1} > 1$. Operația $del(X_{i-1})$ se limitează la eliminarea primului element din R_{i-1}. Lista L_{i-1} nu este modificată.

$$cost(del(X_{i-1})) = 1$$
$$\Phi_i = \#L_i = \#L_{i-1}$$
$$\Phi_{i-1} = \#L_{i-1}$$
$$cost_a(del) = cost(del(X_{i-1})) + \Phi_i - \Phi_{i-1} = 1$$

Caz $\#R_{i-1} = 1$. După eliminarea unicului element din R_{i-1} are loc normalizarea inelului $X' = Abs_p\langle L_{i-1},nil\rangle$ astfel încât $Abs_p\langle L_i,R_i\rangle = Abs_p\langle nil,reverse(L_{i-1})\rangle$. Normalizarea inelului X' costă $\#L_{i-1}$.

$$cost(del(X_{i-1})) = 1 + cost\_normalizare(X') = 1 + \#L_{i-1}$$
$$\Phi_i = \#L_i = \#nil = 0$$
$$\Phi_{i-1} = \#L_{i-1}$$
$$cost_a(del) = cost(del(X_{i-1})) + \Phi_i - \Phi_{i-1} = 1$$

Pentru operațiile $op \in \{top,empty\}$, care nu modifică inelul şi, implicit, conservă potențialul Φ, avem $cost_a(op) = cost(op) = 1$. În ceea ce priveşte cazul particular $void$, se asigură $\Phi(void) = 0$, iar $cost_a(void) = cost(void) = 1$.

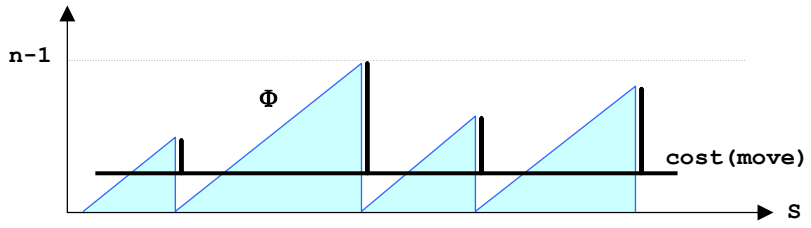

Figura 2.10 Costul real al operației $move$ şi variația potențialului inelului
n este lungimea maximă a inelului în cursul operațiilor s

În figura 2.10 este ilustrată variația potențialului inelului în cursul unei secvențe s de operații asupra unui inel cu cel mult n elemente. Curba Φ este în dinți de fierăstrău. Potențialul Φ creşte pentru fiecare operație $move$ care nu conduce la normalizarea inelului. Pentru o operație $move$ urmată de normalizarea inelului Φ cade

la 0, energia acumulată fiind folosită pentru deplasarea elementelor inelului din lista L în R. Inițial, pentru inelul vid $\Phi=0$.

S-au obținut aceleaşi costuri amortizate ca şi în cazul analizei bazate pe credite. Deoarece $cost_a(op) = O(1)$ pentru orice op∈{void, ins, del, move, top, empty}, costul oricărei secvențe s de m operații pentru inelul unidirecțional (inițial vid) este:

$$cost(S) \le \sum_{i=1}^{m} cost_a(op_i) = O(m)$$

Metoda potențialului a fost aplicată mai sus pentru cazul particular $\Phi(D_0)=0$. Următoarea proprietate arată că analiza conduce la aceleaşi rezultate şi pentru $\Phi(D_0)>0$ cu condiția ca $\Phi(D_i) \ge \Phi(D_0)$, i=1,m.

Propoziția 2.7 Dacă $\Phi(D_0)>0$ şi $\Phi(D_i) \ge \Phi(D_0)$, i=1,m, atunci există o funcție de potențial Φ' astfel încât:

1. $\Phi'(D_0) = 0$ şi $\Phi'(D_i) \ge 0$, i=1,m
2. Costurile amortizate $cost_a'(op_i)$, i=1,m, calculate pentru funcția de potențial Φ', sunt aceleaşi ca şi costurile amortizate $cost_a(op_i)$ calculate pentru Φ.

Să construim Φ' astfel încât $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0)$, i=0,m. Se verifică relațiile de potențial:

$$\Phi'(D_0) = \Phi(D_0) - \Phi(D_0) = 0$$
$$\Phi'(D_i) = \Phi(D_i) - \Phi(D_0) \ge 0$$

$$\begin{aligned} cost_a'(op_i) &= cost(op_i) + \Phi'(D_i) - \Phi'(D_{i-1}) \\ &= cost(op_i) + \Phi(D_i) - \Phi(D_0) - \Phi(D_{i-1}) + \Phi(D_0) \\ &= cost(op_i) + \Phi(D_i) - \Phi(D_{i-1}) \\ &= cost_a(op_i). \end{aligned}$$

■

Pentru funcția $\Phi(Abs_p\langle L,R\rangle) = \#L$, din analiza inelului unidirecțional, propoziția (2.7) nu poate fi aplicată deoarece precondiția $\Phi(D_i) \ge \Phi(D_0)$ necesară aplicării nu este satisfăcută pentru orice i=1,m. Într-adevăr, $\Phi(D_0)>0$ arată că inelul nu este vid la începutul operațiilor din secvența s. Este uşor de observat că după o normalizare efectivă impusă de operatorul op_i rezultă $\Phi(D_i)=0$, iar relația $\Phi(D_i) \ge \Phi(D_0)$ nu este satisfăcută.

În cazuri similare cu cel de mai sus, unde $\Phi(D_0)>0$, $\Phi(D_i)\ge 0$ pentru i=1,m, şi există j∈1..n astfel încât $\Phi(D_j) < \Phi(D_0)$, calculul costurilor amortizate poate fi efectuat considerând o secvență s'

$$S' = D'_0 \text{—} op'_1 \rightarrow D'_1 \text{—} \ldots \rightarrow D'_{m-1} \text{—} op'_m \rightarrow D'_m \text{—} op'_{m+1} \ldots \rightarrow op'_{m+n} \rightarrow D'_{m+n}$$

care satisface restricțiile:

$\Phi(D'_0) = 0$

$D'_{m+i} = D_i$ şi $op'_{m+i} = op_i$, pentru $i=1,n$

$D'_m = D_0$

Secvența s este sufixul $op'_{m+1} \ldots op'_{m+n}$ al secvenței s', secvență în raport cu care funcția de potențial Φ este corectă. Pentru $i=1,n$ avem:

$$\begin{aligned} cost_a(op_i) &= cost_a(op'_{m+1}) \\ &= cost(op'_{m+1}(D'_{m+i-1})) + \Phi(D'_{m+i}) - \Phi(D'_{m+i-1}) \\ &= cost(op_i(D_{i-1})) + \Phi(D_i) - \Phi(D_{i-1}) \end{aligned}$$

Costurile amortizate $cost_a(op_i)$, $i=1,n$, sunt exact cele obținute pornind cu o secvență de operații aplicate unei structuri cu potențial inițial nul şi care, la un moment dat, atinge potențialul nenul Φ_0.

Rezultatul nu este surpinzător, deoarece costul amortizat al unei operații este independent în raport cu secvența de operații care o conține. Astfel, metoda potențialului aplicată unei structuri de date cu potențial inițial nul, este relevantă pentru orice alt caz particular, implict pentru cazurile în care structura de date se află într-o stare inițială validă, dar cu potențial nenul.

În cazul inelului cu starea inițială $D_0 = Abs_p\langle L_0, R_0\rangle$, $L_0 \neq nil$, funcția de potențial $\Phi(D_i) = \#L_i$, $i=0,n$, permite obținerea costurilor amortizate calculate pornind de la un inel inițial vid, care trece prin starea D_0 în cursul prelucrărilor. Aceste costuri amortizate sunt relevante pentru orice secvență de operații şi pentru orice stare inițială a inelului.

2.7.5 Verificarea unor proprietăți ale inelului

Să verificăm proprietatea $\forall i \in \mathbb{N}, X \in T\ Ring \mid \neg empty(X) \bullet Prop(i,X)$, unde

$$Prop(i,X) =_{def} top(move^{i}(X)) = element(i,X)$$

Proprietatea este reprezentativă unui inel unidirecțional şi arată că după `i` deplasări succesive ale vârfului unui inel nevid, elementul de la vârf este al `i`-lea element din inelul inițial. În acest fel, se precizează că deplasarea vârfului păstrează ordinea elementelor din inel. Proprietatea foloseşte următorii operatori auxiliari:

- $element: \mathbb{N} \times (T\ Ring \setminus \{void\}) \to T$ – `element(i,X)` reîntoarce elementul `i mod size(X)` din inelul nevid `X`.

- $elm: \mathbb{N} \times (T\ Ring \setminus \{void\} \to T$ – `elm(i,X)` reîntoarce elementul din poziția `i`, $0 \le i < size(X)$, din inelul nevid `X`.

- $size: T\ Ring \to \mathbb{N}$ – `size(i,X)` este numărul de elemente din inelul `X`. Pentru a scurta scrierea, adoptăm `#X` ca notație alternativă a lui `size(X)`.

Extinsă cu aceşti operatori, specificația tipului `T Ring` devine:

```
T Ring ::= void | ins(T, T Ring)
```

$top: T\ Ring \setminus \{void\} \to T$
$move: T\ Ring \setminus \{void\} \to T\ Ring$
$del: T\ Ring \setminus \{void\} \to T\ Ring$
$empty: T\ Ring \to bool$

$element: \mathbb{N} \times (T\ Ring \setminus \{void\}) \to T$
$elm: \mathbb{N} \times (T\ Ring \setminus \{void\}) \to T$
$size: T\ Ring \to \mathbb{N}$

```
 (1) empty(void) = 1
 (2) empty(ins(e,X)) = 0
 (3) top(ins(e,X)) = e
 (4) del(ins(e,X)) = X
 (5) move(ins(e,void)) = ins(e,void)
 (6) move(ins(e,ins(e',X))) = ins(e',move(ins(e,X)))

 (7) element(i,X) = elm(i mod size(X),X)
 (8) elm(0,X) = top(X)
 (9) elm((i+1),ins(e,X)) = elm(i,X), pentru i < size(X)
(10) size(void) = 0
(11) size(ins(e,X)) = 1+size(X)
```

Pentru verificarea `Prop(i,X)`, sunt utile următoarele două propoziții, demonstrate prin inducție structurală.

Propoziția 2.8 $\forall X \in T\ Ring \mid \neg empty(X) \bullet P_1(X)$, cu $P_1(X) =_{def} \#move(X) = \#X$

Caz de bază. `X = ins(e,void)`.

```
#move(ins(e,void)) —— 5 → #ins(e,void)
```

Pas de inducție. `X = ins(e,ins(e',X'))`. Ca ipoteză inductivă considerăm adevărată proprietatea $\forall e \in T \bullet P_1(ins(e,X'))$.

```
#move(ins(e,ins(e',X')))
   —— 6 → #ins(e',move(ins(e,X')))
   —— 11 → 1+ #move(ins(e,X'))
   —— ip.ind → 1+#ins(e,X')
   —— 11 → 2+#X'

#ins(e,ins(e',X'))
   —— 11 → 1+#ins(e',X')
   —— 11 → 2+#X'
■
```

Propoziția 2.9 $\forall i \in \mathbb{N}, X \in T\ Ring \mid \neg empty(X) \bullet P_2(i,X)$, unde

$$P_2(i,X) =_{def} element(i,move(X)) = element(i+1,X)$$

Caz de bază. `X = ins(e,void)`. Să verificăm $P_2(i,X)$, pentru $i \in \mathbb{N}$ ales arbitrar.

```
element(i,move(ins(e,void)))
   —— 5 → element(i,ins(e,void))
   —— 7 → elm(i mod #ins(e,void),ins(e,void))
   —— 11,10 → elm(i mod 1, ins(e,void))
   —— mod → elm(0, ins(e,void))
   —— 8,3 → e

element(i+1,X)
   —— 7 → elm((i+1) mod #ins(e,void),ins(e,void))
   —— 11,10 → elm((i+1) mod 1, ins(e,void))
   —— mod → elm(0, ins(e,void))
   —— 8,3 → e
```

Pas de inducție. `X = ins(e,ins(e',X'))`. Ca ipoteză inductivă considerăm că este adevărată proprietatea $\forall e \in T \bullet P_2(i,ins(e,X'))$, pentru $i \in \mathbb{N}$ ales arbitrar. Să arătăm că proprietatea $P_2(i,X)$ este adevărată. Notăm:

```
LHS = element(i,move(X))
RHS = element(i+1,X)

LHS = element(i,move(ins(e,ins(e',X'))))
   —— 6 → element(i,ins(e',move(ins(e,X'))))
   —— 7,11,P1 → elm(i mod (2+#X'),ins(e',move(ins(e,X'))))
```

Caz (a): `i mod (2+#X')=0`. Atunci, avem `(i+1) mod (2+#X') = 1`.

```
LHS = elm(0,ins(e',move(ins(e,X')))) ⟶ 8,3 → e'

RHS = element(i+1,ins(e,ins(e',X')))
   ⟶ 7,11 → elm((i+1) mod (2+#X'), ins(e,ins(e',X')))
   ⟶ [(i+1) mod (2+#X') = 1] → elm(1, ins(e,ins(e',X')))
   ⟶ 9 → elm(0,ins(e'X'))
   ⟶ 8,3 → e'
```

Caz (b): `i mod (2+#X')>0`. Atunci, `i>0` şi `i mod (2+#X')= (i-1) mod (2+#X')+1`.

```
LHS = elm((i-1) mod (2+#X')+1,ins(e',move(ins(e,X'))))
   ⟶ 9 → elm((i-1) mod (2+#X'),move(ins(e,X')))
   ⟶ [(i-1) mod (2+#X') < 1+#X'] →
         elm(((i-1) mod (2+#X')) mod(1+#X'), move(ins(e,X')))
   ⟶ P1,7 → element((i-1) mod (2+#X'), move(ins(e,X')))
   ⟶ ip.ind → element((i-1) mod (2+#X')+1, ins(e,X'))
   ⟶ [i mod (2+#X') = (i-1) mod (2+#X')+1] →
            element(i mod (2+#X'), ins(e,X'))
```

Caz (b1): `i mod (2+#X') = 1+#X'`

```
LHS = element(1+#X', ins(e,X'))
   ⟶  7 →  elm((1+#X') mod ins(e,X'), ins(e,X'))
   ⟶ 11 →  elm((1+#X') mod (1+#X'), ins(e,X'))
   ⟶ mod → elm(0, ins(e,X'))
   ⟶ 8,3 → e
```

Totodată, `i mod (2+#X') = 1+#X'` ⇒ `(i+1) mod (2+#X') = 0`

```
RHS = element(i+1, ins(e,ins(e',X')))
   ⟶ 7,11 → elm((i+1) mod (2+#X'), ins(e,ins(e',X')))
   ⟶ [(i+1) mod (2+#X') = 0] → elm(0, ins(e,ins(e',X')))
   ⟶ 8,3 → e
```

Caz (b2): `i mod (2+#X') < 1+#X'`

```
LHS = element(i mod (2+#X'), ins(e,X'))
   ⟶ 7,11 → elm((i mod (2+#X')) mod (1+#X'), ins(e,X'))
   ⟶  mod → elm(i mod (2+#X'), ins(e,X'))
   ⟶ 9,[0 < i mod (2+#X') < 1+#X'] → elm(i mod (2+#X')-1, X')

RHS = element(i+1, ins(e,ins(e',X')))
   ⟶ 7,11 → elm((i+1) mod (2+#X'), ins(e,(ins(e',X')))
   ⟶ [(i+1) mod (2+#X') = (i mod (2+#X'))+1] →
            elm((i mod (2+#X'))+1, ins(e,(ins(e',X')))
   ⟶ 9 → elm(i mod (2+#X'), ins(e',X'))
   ⟶ 9,[0 < i mod (2+#X') < 1+#X'] → elm(i mod (2+#X')-1, X')
```

■

Propoziția 2.10 Există proprietatea $\forall i \in \mathbb{N}, X \in T\ Ring\ |\neg empty(X) \bullet Prop(i,X)$, unde

$$Prop(i,X) =_{def} top(move^{i}(X)) = element(i,X).$$

Demonstrația este prin inducție după `i`.

Caz de bază. `i=0`.

$top(move^{0}(X)) = top(X)$

$element(0,X)$
- $7 \rightarrow elm(0 \bmod \#X, X)$
- $mod \rightarrow elm(0,X)$
- $8 \rightarrow top(X)$

Pas de inducție. $i \geq 0$. Ipoteza inductivă este $\forall X \in T\ Ring | \neg empty(X) \bullet Prop(i,X)$. Să arătăm că $Prop(i+1,X)$ este adevărată, adică

$$top(move^{i+1}(X)) = element(i+1,X)$$

$top(move^{i+1}(X)) = top(move^{i}(move(X)))$
- $ip.ind \rightarrow element(i, move(X))$
- $P_2 \rightarrow element(i+1,X)$

■

2.7.6 O implementare Haskell a inelului

Deşi codificarea directă – într-un limbaj de programare funcțională – a specificațiilor tipurilor de date şi implementărilor abstracte depăşeşte cadrul acestei cărți, o incursiune superficială şi succintă în acest domeniu este interesantă. Este scopul acestei secțiuni care prezintă:

- Codificarea Haskell [Tho 96] a specificației `T Ring`.
- Implementarea efectivă a inelului conform $T\ Ring = Abs\langle T\ List, T\ List\rangle$.
- O aplicație a inelului pentru sortarea unei liste de chei.

Programele de mai jos corespund aproape ad-litteram specificației `T Ring` sau ecuațiilor de implementare şi, evident, sunt executabile. În particular, programul corespunzător specificației `T Ring` poate fi considerat ca o implementare "abstractă" a inelului. Totuşi, în programele Haskell apar următoarele diferențe minore sau variante de notație în raport cu textul din carte:

Program Haskell	Specificație `T Ring`
`Ring t`	`T Ring`
`e:l`	`cons(e,l)`
`[]`	`nil` (lista vidă)
`(f x)`	`f(x)`

Rescrierea Haskell a specificației inelului

```
module T_Ring where

-- Constructorii de bază: Void şi Ins
-- Limbajul impune nume ce încep cu litere mari

data Ring t = Void
            | Ins (t, Ring t) deriving (Eq,Show)

-- Funcții corespunzătoare constructorilor de bază, utile
-- pentru compatibilitate deplină cu implementarea inelului

void = Void
ins(e,x) = Ins(e,x)

-- Axiomele operatorilor (signaturile sunt sintetizate automat)

empty(Void) = True
empty(Ins(_,_)) = False
top(Ins(e,_)) = e
del(Ins(_,x)) = x

move(Ins(e,Void)) = Ins(e,Void)
move(Ins(e,Ins(e',x))) = Ins(e',move(Ins(e,x)))

-- Axiomele operatorilor necesari proprietăților P1,P2 şi Prop

element(i,x) = elm(i `mod` size(x), x)
elm(0,x) = top(x)
elm(i+1,Ins(_,x)) = elm(i,x)

size(Void) = 0
size(Ins(_,x)) = 1+size(x)

-- Proprietatile P1, P2 şi Prop din secțiunea 2.7.5, unde
-- move_i(n,x) este move aplicat de n ori inelului x

p1 x = size(x) == size(move(x))
p2(i,x) = element(i,move(x)) == element(i+1,x)
prop(i,x) = top(move_i(i,x)) == element(i,x)
            where move_i(0,x) = x
                  move_i(i+1,x) = move_i(i,move(x))

-- Funcție care construieşte un inel cu cheile unei liste
make_ring [] = Void
make_ring (e:l) = Ins(e,make_ring l)

-- Un inel
x = make_ring [1,2,3,4,5,6,7,8]
```

Aplicațiile `p1(x)`, `p2(i,x)` şi `prop(i,x)` reîntorc valoarea `True` pentru orice `i≥0` si orice inel nevid `x`.

Codificarea Haskell a implementării inelului

```
module Ring where
-- O implementare a tipului T Ring in Haskell, ca o pereche de liste

-- Reprezentarea inelului ca pereche de liste cu elemente de tip t
data Ring t = Abs([t],[t]) deriving (Eq,Show)

normalize(l,r) = if r==[] then ([],(reverse l)) else (l,r)

-- Ecuațiile de implementare a operatorilor inelului
void = Abs([],[])
ins (e,Abs(l,r)) = Abs(l,e:r)
del (Abs(l,r)) = Abs(normalize(l,(tail r)))
move (Abs(l,r)) = Abs(normalize((head r):l, (tail r)))
top (Abs(l,r)) = head r
empty (Abs(l,r)) = r == []

-- Axiomele operatorilor necesari proprietăților P1,P2 şi Prop

size(x) = if empty(x) then 0 else 1+size(del(x));;
elm(0,x) = top(x)
elm(i+1,x) = elm(i,del(x))
element(i,x) = elm(i `mod` size(x), x)

{-
   Axioma elm(i+1,ins(e,x))) = elm(i,x) este rescrisa în stilul
   elm(i+1,y) = elm(i,del(y)) folosind schimbarea de variabilă
   y = ins(e,x) => x = del(y).

   La fel s-a procedat şi pentru axioma size(ins(e,x)) = 1+size(x)
   scrisă size(x) = 1+size(del(x))
-}

-- Proprietatile P1, P2 şi Prop din secțiunea 2.7.5, unde
-- move_i(i,x) este move aplicat de i ori inelului x

p1 x = size(x) == size(move(x))
p2(i,x) = element(i, move(x)) == element(i+1,x)
prop(i,x) = top(move_i(i,x)) == element(i,x)

move_i(0,x) = x
move_i(i+1,x) = move_i(i,move(x))

-- Funcție care construieşte un inel cu cheile unei liste
make_ring [] = void
make_ring (e:l) = ins(e,make_ring l)

-- Un inel
x = make_ring [1,2,3,4,5,6,7,8]
```

Aplicațiile `p1(x)`, `p2(i,x)` şi `prop(i,x)` reîntorc valoarea `True` pentru orice `i≥0` şi orice inel nevid `x`. Evident, din moment ce implementarea respectă axiomele specificației `T Ring`, ea conservă şi proprietățile inelului.

O aplicație: sortare pe bază de inel

Sortarea unei liste unidirecționale folosind un inel de tip `T Ring` se aseamănă cu sortarea prin inserție directă a unui vector. Pentru simplitate, considerăm sortarea nedescrescătoare a unei liste cu chei întregi, deci presupunem că tipul generic `T` din specificațiile `T List` şi `T Ring` este particularizat la tipul de date `int`.

O variantă simplă de sortare a unei liste nevide `l` constă în construirea unui nou inel sortat prin inserția elementului `head(l)` într-un inel sortat în aceeaşi manieră şi care conține elementele listei `tail(l)`. Evident, pentru o listă vidă, inelul sortat este vid. Operația de bază a sortării este

```
put: int × int Ring → int Ring
```

şi satisface axiomele

```
(p1) put(e,void) = ins(e,void)
(p2) put(e,ins(e',X)) = if e < e' then ins(e,ins(e',X))
                                  else ins(e',put(e,X))
```

Să arătăm că, pentru `X∈int Ring` deja sortat nedescrescător, `put(e,X)` produce un inel sortat nedescrescător. Pentru că avem nevoie de formalizarea noțiunii de inel sortat nedescrescător, definim următoarele predicate:

`ord: int Ring → bool` şi
`min: int × (int Ring) → bool`

astfel încât:

```
(o1) ord(void) = 1
(o2) ord(ins(e,X)) = min(e,X) ∧ ord(X)

(m1) min(e,void) = 1
(m2) min(e,ins(e',X)) = e ≤ e' ∧ min(e,X)
```

De asemenea, considerăm adevărate proprietățile

```
∀X∈int Ring, e∈int,e'∈int • P'(e,e',X)
∀X∈int Ring, e∈int,e'∈int • P"(e,e',X)
```

unde,

```
P'(e,e',X) =def min(e,put(e',X)) = e ≤ e' ∧ min(e,X)
P"(e,e',X) =def e ≤ e' ∧ min(e',X) ⇒ min(e,X)
```

Demonstrațiile sunt prin inducție structurală, fiind lăsate ca exercițiu.

Folosind `ord`, proprietatea esențială pe care se bazează procesul de sortare poate fi formulată `∀X∈int Ring, e∈int • P(e,X)`, cu

```
P(e,X) =def ord(X) ⇒ ord(put(e,X))
```

Propoziția 2.11 `∀X∈int Ring • P(X)`. Demonstrația este prin inducție structurală.

Caz de bază: `X = void`. Să alegem `e∈T` arbitrar.

```
P(e,void) = ord(void) ⇒ ord(put(e,void))
        — o1 → 1 ⇒ ord(put(e,void))
        — p1 → 1 ⇒ ord(ins(e,void))
        — o2 → 1 ⇒ min(e,void) ∧ ord(void)
        — m1,o1 → 1 ⇒ 1
        = 1
```

Pas de inducție: `X = ins(e',X')`, unde `e'∈int` şi `X'∈int Ring` sunt valori alese arbitrar. Ca ipoteză inductivă, considerăm adevărată proprietatea `P(e,X')` pentru `e∈int` ales arbitrar. Să arătăm că proprietatea `P(e,X)` este adevărată, anume că:

```
ord(ins(e',X')) ⇒ ord(put(e,ins(e',X')))
```

Dacă termenul `ord(ins(e',X'))` este fals, implicația este adevărată banal. Să considerăm că `ord(ins(e',X'))` este adevărat. Atunci, conform axiomei `(o2)` avem următoarele concluzii:

```
(c1) min(e',X') = 1
(c2) ord(X') = 1
```

Pentru ca implicația `P(e,X)` să fie adevărată trebuie ca termenul `ord(put(e,ins(e',X')))` să fie adevărat.

Caz 1. `e < e'`

```
ord(put(e,ins(e',X')))
       — p2 → ord(ins(e,ins(e',X')))
       — o2 → min(e,ins(e',X')) ∧ ord(ins(e',X'))
       — m2 → e ≤ e' ∧ min(e,X') ∧ ord(ins(e',X'))
       — Caz 1, c1, P"(e,e',X')→ ord(ins(e',X'))[1]
       — ipoteză → 1
```

Caz 2. `e ≥ e'`

```
ord(put(e,ins(e',X')))
       — p2 → ord(ins(e',put(e,X')))
       — o2 → min(e',put(e,X')) ∧ ord(put(e,X'))
       — ip.induct. P(e,X'), c2 → min(e',put(e,X'))[2]
       — P'(e',e,X') → e' ≤ e ∧ min(e',X')
       — Caz 2, c1 → 1
```

■

---

[1] Termenul `min(e,X')` este adevărat conform implicației `P"(e,e',X')`

[2] Termenul `ord(put(e,X'))` este adevărat conform implicației `P(e,X')`

Considerând lista de sortat `l = cons(l1,cons(l2,...,cons(ln,nil)...))`, atunci, prin inducție după `n` şi conform propoziției (2.11), inelul

`r = put(l1,put(l2,...,put(ln,void)...)`

rezultă sortat nedescrescător. Funcția care care construieşte inelul `r` este:

```
ring_sort: int List → int Ring

ring_sort(nil) = void
ring_sort(cons(e,L)) = put(e,ring_sort(L))
```

Dacă dorim ca rezultatul sortării să fie o listă, inventăm o funcție care descarcă elementele unui inel (indiferent de tipul elementelor) într-o listă inițial vidă.

```
ring_to_list: T Ring → T List

ring_to_list(void) = nil
ring_to_list(ins(e,X)) = cons(e,ring_to_list(X))
```

În final, funcția de sortare este reprezentată de compunerea funcțiilor `ring_sort` şi `ring_to_list`:

```
sort: int List → int List

sort L = (ring_to_list o ring_sort)(L)
       = ring_to_list(ring_sort(L))
```

Transcrierea Haskell a sortării, impune modificarea axiomelor funcțiilor `put` şi `ring_to_list`.[3] Prima axiomă a funcției `put` este transformată prin schimbarea de variabilă `x = void`, impunând restricția `empty(x)`. Cea de a doua axiomă a lui `put` este transformată folosind schimbarea de variabilă `x = ins(e',X)` şi observând, pe baza axiomelor din specificația inelului, că: `e' = top(x)` şi `X = del(x)`. Se obține:

```
          | ins(e,void), dacă empty(X)
put(e,X) = | ins(e,X), dacă ¬empty(X) ∧ e < top(X)
          | ins(top(X),put(e,del(X))), dacă ¬empty(X) ∧ e ≥ top(X)
```

Similar, sunt transformate şi axiomele funcției `ring_to_list`:

```
                    | nil, dacă empty(X)
ring_to_list(X) =   |
                    | cons(top(X),ring_to_list(del(X)))
```

[3] Motivul îl constituie definirea constructorilor `void` şi `ins` ca funcții implementate pe baza constructorului `Abs`. Haskell nu acceptă ca parametrii formali ai funcțiilor să fie specificați în forma unor aplicații de funcții, aşa cum cer axiomele nemodificate ale lui `put` şi `ring_to_list`. Totodată, codificarea din modulul `RingSort` are un avantaj: funcționează corect cu ambele implementări ale inelului, conforme modulelor `T_Ring` şi `RingImpl`.

Într-o primă variantă, codificarea Haskell a sortării urmăreşte îndeaproape specificarea funcției **sort**.

```
module RingSort where
import RingImpl

sort l = (ring_to_list . ring_sort) l
         where -- f . g este compunerea lui f cu g
               ring_sort [] = void
               ring_sort (e:l) = put(e,ring_sort(l))

               put(e,x)
                 | empty(x) = ins(e,void)
                 | e < top(x) = ins(e,x)
                 | otherwise = ins(top(x),put(e,del(x)))

               ring_to_list x
                 | empty(x) = []
                 | otherwise = (top x):(ring_to_list (del x))
```

O variantă mai sofisticată foloseşte o prelucrare interesantă, dar comună în limbajele de programare funcționale. Prelucrarea corspunde următoarei funcții:

```
foldr: ((T × T') → T') × T' × (T List) → T'
foldr(f,X,nil) = X
foldr(f,X,cons(e,L)) = f(e,foldr(f,X,L))
```

Prin urmare, aplicația **foldr(put,void,l)**, unde **l = cons(l_1,cons(l_2,...,cons(l_n,nil)...))**, este echivalentă cu **put(l_1,put(l_2,...,put(l_n,void)...)**, adică are efectul **ring_sort(l)**.

```
module RingSort where
import RingImpl

sort l = ring_to_list(foldr put void l)
       where
              put e x  -- vezi comentariul[4]
                | empty(x) = ins(e,void)
                | e < top(x) = ins(e,x)
                | otherwise = ins(top(x), put e (del x))

              ring_to_list x
                | empty(x) = []
                | otherwise = (top x):(ring_to_list (del x))
```

[4] **foldr** cere ca **put** să fie o funcție binară. În Haskell, **put(e,x)** defineşte o funcție cu un singur parametru din mulțimea produs **int** × **int Ring**, în timp ce **put e x** descrie o funcție binară cu primul parametru de tip **int** şi al doilea parametru de tip **int Ring**. Diferența este mai subtilă: o funcție binară poate fi aplicată parțial. Aplicarea asupra primului parametru produce o funcție unară care aşteaptă cel de al doilea parametru. Din program, se vede că însăşi **foldr** este o funcție ternară care are proprietăți similare.

La încărcarea şi execuția programului, se poate obține un rezultat de forma:

```
RingSort> sort [5,1,2,7,1,2,5,6,4]
[1,1,2,2,4,5,5,6,7] :: [Integer]
```

Modulele Haskell de mai sus implementează mai mult decât sortarea unei liste de întregi. Ele descriu sortarea unei liste cu elemente de orice tip peste care este definită o relație de ordine. Generalitatea este vizibilă din signatura Haskell a funcției `sort`, sintetizată automat, anume:

```
RingSort> :type sort
sort :: Ord t => [t] -> [t]
```

Signatura arată că tipul `t` trebuie să facă parte din clasa `Ord`, clasa tipurilor peste care este definită o relație de ordine. Astfel, se poate scrie:

```
RingSort> sort [True,False]
[False,True] :: [Bool]
RingSort> sort [2.1,3.0,3.14,0.1,4.5]
[0.1,2.1,3.0,3.14,4.5] :: [Double]
```

Complexitatea amortizată a sortării, anume $C(n) = T(n) + D(n) + R(n)$, unde n este numărul de chei din lista de sortat, ține cont de complexitatea $T(n)$ a comparării cheilor, de complexitatea amortizată $D(n)$ a operațiilor `del` efectuate în cursul sortării şi de complexitatea amortizată $R(n)$ a descărcării inelului sortat în lista rezultat. La inserția celei de a i-a chei în inel se efectuează cel mult $i-1$ comparații, iar lungimea secvenței de operații `del` este i. Descărcarea inelului implică o secvență de n operații `del`, celelalte operații ale descărcării având complexitate $O(1)$. Deci,

$$T(n) \le \Theta(1) \sum_{i=1}^{n}(i-1),\ D(n) = \sum_{i=1}^{n} O(i) \text{ şi } R(n) = O(n)$$

Rezultă $C(n) = O(n^2)$.

Comentarii

Conținutul secțiunii 2.7.6 depăşeşte cadrul strict al analizei algoritmilor, fiind la intersecția următoarelor domenii: metode formale, programare funcțională, analiza algoritmilor. Dar morala este vizibilă şi importantă: este mai sigur şi, în unele cazuri, mai uşor să se dezvolte teoretic un algoritm corect decât să fie construit un algoritm "intuitiv" care, ulterior, trebuie verificat. Totodată, dezvoltarea elaborată, pornind de la specificații, poate salva timp şi efort în procesul implementării algoritmului şi validării programului. Concluzia capătă greutate mai ales pentru aplicațiile critice, acolo unde o anomalie a algoritmului poate fi extrem de costisitoare.

Cine doreşte să afle mai mult despre Haskell poate consulta situl http://www.haskell.org, unde se găsesc medii de programare Haskell, inclusiv documentație Haskell (manual de referință).

Exerciții

Propoziția 2.12 $\forall X \in int\ Ring,\ e \in int, e' \in int \bullet P'(e,e',X)$, unde

$$P'(e,e',X) =_{def} min(e,put(e',X)) = e \leq e' \wedge min(e,X)$$

Demonstrația este prin inducție structurală. Alegem $e \in T$, $e' \in T$ şi $X \in T\ Ring$ la întâmplare şi notăm:

$$LHS = min(e,put(e',X))$$
$$RHS = e \leq e' \wedge min(e,X)$$

Caz de bază. $X = void$.

$$LHS = min(e,put(e',void))$$
$$\quad \text{— p2} \rightarrow e \leq e' \wedge min(e,void)$$
$$\quad \text{— m1} \rightarrow e \leq e'$$

$$RHS = e \leq e' \wedge min(e,void)$$
$$\quad \text{— m1} \rightarrow e \leq e' = LHS$$

Pas de inducție. Alegem ca ipoteză inductivă $P'(e,e',X)$ şi demonstrăm că proprietatea $P'(e,e',ins(e",X))$ este adevărată, pentru $e" \in T$ ales la întâmplare.

$$RHS = e \leq e' \wedge min(e,ins(e",X))$$
$$\quad \text{— m2} \rightarrow e \leq e' \wedge e \leq e" \wedge min(e,X)$$

Caz 1. $e' < e"$

$$LHS = min(e,put(e',ins(e",X)))$$
$$\quad \text{— p2} \rightarrow min(e,ins(e',ins(e",X)))$$
$$\quad \text{— m2} \rightarrow e \leq e' \wedge min(e,ins(e",X))$$
$$\quad \text{— m2} \rightarrow e \leq e' \wedge e \leq e" \wedge min(e,X)$$
$$\quad \text{— Caz 1} \rightarrow e \leq e' \wedge min(e,X)$$

$$RHS = e \leq e' \wedge min(e,ins(e",X))$$
$$\quad \text{— m2} \rightarrow e \leq e' \wedge e \leq e" \wedge min(e,X)$$
$$\quad \text{— Caz 1} \rightarrow e \leq e' \wedge min(e,X) = LHS$$

Caz 2. $e' \geq e"$

$$LHS = min(e,put(e',ins(e",X)))$$
$$\quad \text{— p2} \rightarrow min(e,ins(e",put(e',X)))$$
$$\quad \text{— m2} \rightarrow e \leq e" \wedge min(e,put(e',X))$$
$$\quad \text{— ip.iduct.} \rightarrow e \leq e" \wedge e \leq e' \wedge min(e,X)$$
$$\quad \text{— Caz 2} \rightarrow e \leq e" \wedge min(e,X)$$

$$RHS = e \leq e' \wedge min(e,ins(e",X))$$
$$\quad \text{— m2} \rightarrow e \leq e' \wedge e \leq e" \wedge min(e,X)$$
$$\quad \text{— Caz 2} \rightarrow e \leq e" \wedge min(e,X) = LHS$$

■

Propoziția 2.13 $\forall X \in$ `int Ring, e∈int,e'∈int • P"(e,e',X)`, unde

$$\texttt{P"(e,e',X)} =_{\texttt{def}} \texttt{e} \leq \texttt{e'} \wedge \texttt{min(e',X)} \Rightarrow \texttt{min(e,X)}$$

Demonstrația este prin inducție structurală. Alegem `e∈T`, `e'∈T` şi `X∈T Ring` la întâmplare şi notăm:

```
LHS = e ≤ e' ∧ min(e',X)
RHS = min(e,X)
```

Caz de bază. `X = void`.

```
LHS = e ≤ e' ∧ min(e',void)
        — m1 → e ≤ e'

RHS = min(e,void)
        — m1 → 1
```

S-a obținut tautologia `e ≤ e' ⇒ 1`.

Pas de inducție. Alegem ca ipoteză inductivă `P"(e,e',X)` şi demonstrăm că proprietatea `P"(e,e',ins(e",X))` este adevărată, pentru `e"∈T` ales la întâmplare.

```
LHS = e ≤ e' ∧ min(e',ins(e",X))
        — m2 → e ≤ e' ∧ e' ≤ e" ∧ min(e',X)
```

Să considerăm că `LHS = 1`. Atunci,

`(c1)` Conform ipotezei inductive, `min(e,X) = 1` şi
`(c2)` `e ≤ e"`

```
RHS = min(e,ins(e",X))
        — m2 → e ≤ e" ∧ min(e,X)
        — c1,c2 → 1
```

■