De rij van unieke bladwaarden van een boom

Maarten Fokkinga, 27 mei 1986

mf verbeterbaar

We behandelen het probleem om bij gegeven willekeurige boom de rij van die bladwaarden op te leveren die precies éénmaal in de boom (als bladwaarde) voorkomen.

* * *

Zonder verlies van algemeenheid stellen we dat bomen gedefinieerd zijn volgens het type

$$T * \;::=\; A * \;\mid\; C\ (T *)\ (T *)$$

Hierin staat A voor Atom en C voor Constructie-van. Gevraagd wordt een functie f zó dat f t = de lijst van bladwaarden (Atoms) die precies éénmaal in t voorkomen (als Atom, dus) en wel zo dat de volgorde der elementen in (f t) gelijk is aan die in t. Een formele specificatie voor f luidt bijvoorbeeld als volgt.

Er moet gelden $\quad sub\ (f\,t,\ \underbrace{t}_{\text{leaves}}) \quad$ en $\quad \forall x \in f\,t.\ \#_x t = 1$

en $\forall x \in t.\ \#_x t = 1 \rightarrow x \in f\,t$

waarbij

$sub\ ([\,],\ X) \leftarrow true.$

$sub\ (X +\!\!+ [y] +\!\!+ Z,\ X' +\!\!+ [y] +\!\!+ Z') \leftarrow \cancel{sub(X +\!\!+ Z,\ X' +\!\!+ Z')}$ $sub\ X\ Z \wedge sub\ X'\ Z'$

en

$leaves\ A x = [x]$
$leaves\ (C\,t\,t') = leaves\ t +\!\!+ leaves\ t'$

$$\#_x (A\ y) = 0\ ,\quad y \neq x$$
$$= 1\ ,\quad y = x$$
$$\#_x (C\ t\ t') = \#_x t + \#_x t'$$

(Hierbij is sub gespecificeerd middels Horn clause logic (Prolog), en $\#_x$ middels een functioneel programma.)

De eerste stap in de programma-ontwikkeling is om te proberen e'n inductieve definitie voor f samen te stellen. Zonder aan te geven hoe we uit de specificatie tot hetvolgende zijn gekomen, geven we hier het resultaat van die poging.

$$f\ (A\ x) = [x]$$
$$f\ (C\ t\ t') = (f\ t \;--\; g\ t') \;+\!\!+\; (f\ t' \;--\; g\ t)$$

waarbij $g\ t$ = een lijst van alle bladwaarden van t, en de formeler, waarbij g is gespecificeerd door

$$(\forall x \in g\,t.\ \ x \in t) \ \wedge\ (\forall x \in t.\ \ x \in g\,t)$$

We moeten dus ook g uit programmeren. Als eerste stap proberen we ook voor g een inductieve definitie af te leiden uit de (in)formele specificatie. Dat geeft:

```
g (A x)    = [x]
g (C t t') = g t ++ g t'
```

De tweede stap in de programma-ontwikkeling is nu om f en g tot één functie h te combineren. De efficientie-verbetering hiervan is tweeledig. Enerzijds voorkomen we dat de bomen tweémaal aan een ontleding (inductie) worden onderworpen. Anderzijds, en dit is véél belangrijker, voorkomen we veelvuldige berekening van (g t) voor identieke t-waarden. Immers, bekijk eens (f t, g t):

```
(f (A x), g (A x))       = ([x], [x])
(f (C t t'), g (C t t')) = ((f t -- g t') ++ (f t' -- g t),  g t ++ g t')
```

We zien hierbij dat de berekeningen van (g t) en (g t') tweémaal worden opgeroepen.[F] Dit wordt voorkomen door de functie h met h t = (f t, g t) gedefinieerd als volgt:

```
h (A x)    = ([x], [x])
h (C t t') = ((ft -- gt') ++ (ft' -- gt),  gt ++ gt')
             where  (ft, gt)   = h t
                    (ft', gt') = h t'
```

F Omdat g recursief is, versterkt dit proces zich: zij t een subboom op diepte n van het oorspronkelijke argument, dan wordt de berekening van (g t) 2^n maal opgeroepen.

(Er is overigens ook een andere manier om de veelvuldige oproep van identieke berekeningen te voorkomen: lazy memoisation. Door alles onveranderd te laten ~~behalve~~ en alleen het directief lazymemo g toe te voegen, worden berekeningen op identieke argumenten niet herhaald maar opgezocht in een tabel van argument-resultaat paren. Zie [Hughes 1985].)

De derde stap in de programma-ontwikkeling is de eliminatie van de linksnesting van ++. We passen de motivatie en techniek van [Fokkinga 1986] toe, en vinden

```
h' t = j t ([],[])
       where
       j (A x) (ft', gt') = (x: ft', x: gt') ?   , singleton_x
                          = (ft' -- [x], x: gt') , otherwise
                            where
                            singleton_x = and [y≠x | y<-ft'] &
                                          and [y≠x | y<-gt']
       j (C t1 t2) (ft', gt') = j t1 (j t2 (ft', gt'))
```

De functie j voldoet aan j t (ft', gt') = h (C t t') = (f (C t t'), g (C t t')). We kunnen nu definiëren

```
f t = ft where (ft, gt) = h' t  ||= j t ([],[])
```

We kunnen de efficientie nog iets opvoeren door in de functie j de test of x in ft' voorkomt te vermengen met het verwijderen van x uit ft' (in de otherwise - clause). Maar de grootte-orde van de tijdscompleciteit verandert hierdoor niet: die blijft kwadratisch in het aantal bladeren van de boom. (Immers, voor ieder blad Ax moet singleton-x berekend worden; dit kost, gesommeerd over alle bladeren, ~~ongeveer~~ een aantal vergelijkingen dat ligt in de orde van grootte van n, met n= aantal bladeren in de boom.)

Bovenstaande efficientie overweging suggereert onmiddellijk nog een alternatieve oplossing. Héél vaak is het zo dat voor gesorteerde rijen een oplossing lineair is in de lengte van de rij, terwijl er sorteermethoden zijn die n log n - stappen kosten met n= lengte van de rij. Dus sorteren-en-dan-triviaal-oplossen kost n + n log n stappen, en dat is minder dan n^2.

Zonder enig probleem vinden we de volgende oplossing

```
f = single • quicksort • flatten
    where single [] = []
          single[x] = [x]
          single(x:y:Z) = x: single(y:Z) , x ≠ y
                        = single(y:Z) , x = y
```

Fout: oorspronkelijke volgorde gaat verloren

Beter: ~~f t = single [] (flatten t)~~
~~met~~
~~single Y [] = []~~
~~single Y (x:X) = x: single (x:Y) X, x ∉ Y~~
~~= single Y X, x ∈ Y~~

```
f t = single (flatten t)
    where single (x:X) = x : single X ,  x ∉ X
                       = (single X) -- [x],  x ∈ X   (of: single (X -- [x]), x ∈ X )
```

Zorg nu dat single tweetal oplevert: s' X = (single X, set X) zo dat test x ∈ X wordt: x ∈ set X. (Effic).

Hiermee is onze programma-ontwikkeling voltooid. Hetzelfde probleem wordt ook behandeld in [Henderson 1981].

Literatuur

~~Fokkinga, M.M,~~ Eliminatie van lijstnesting. Manuscript, 27 mei 1986.

Henderson, P., Functional Programming - Application and Implementation, Prentice-Hall, 1981

Hughes, J., Lazy memo-functions. In Functional programming languages and Computer Achitecture, (ed. JP Jouannaud), Springer-Verlag, LNCS 201 (1985) pp 129-146.

Fokkinga, M.M., Elimination of list nesting - an example of the style of functional programming. Memorandum INF-86-??, T.H. Twente, 1986