Optellen en vermenigvuldigen in Bird's notatie

Maarten Fokkinga, 29 april 1987.

We beschouwen de lagere school-algoritmen voor het optellen en vermenigvuldigen als gegeven, en drukken die uit in Bird's notatie. Dit doen we enerzijds zónder en anderzijds mét expliciete recursie. De formulering met expliciete recursie dwingt af dat we in de decimale representatie van getallen de eenheden "op hop" hiezen; dit is bij de "recursie-loze" formulering niet nodig.

De gegeven algoritme voor optelling

In dit verhaal identificeren we <u>cijfers</u> met hun <u>getalwaarden</u>. Getallen worden gerepresenteerd door hun decimale notatie: de cijferrij die de eenheden, tientallen, honderdtallen etc van het getal aangeeft. Daarbij kunnen we dan nog zo'n cijferrij van links naar rechts, of van rechts naar links in een lijst weergeven:

Een getal $n = \sum_{i=0}^{k} x_i \times 10^i$ met $0 \leq x_i \leq 9$ $(i = 0..k)$ wordt voorgesteld door

- $[x_k, \ldots, x_1, x_0]$ ("vrnl - representatie")
- $[x_0, x_1, \ldots, x_k]$ ("vlnr - representatie")

We proberen de lagere school algoritmen zo getrouw mogelijk weer te geven, en kiezen daarom de vnl-representatie om mee te beginnen. In feite is dit nogal naief: je kunt je beter niet te vroeg op representatie-keuzen vastpinnen. Het zal blijken dat we bij gebruik van expliciete recursie de vlnr-representatie, met de eenheden op kop, moeten kiezen!

De essentie van de algoritme voor optelling luidt als volgt; zie ook Figuur 1.

$$
\begin{array}{rcccl}
 & c' & c & \cdots & c_1 \quad (c_0 = 0) \\
x_m & \cdots & x & \cdots & x_1 \quad x_0 \\
y_n & \cdots & y & \cdots & y_1 \quad y_0 \\
\hline
 & & z & \cdots & z_1 \quad z_0
\end{array}
\qquad +
$$

Figuur 1. Optelling van getallen uitgedrukt in bewerkingen op de decimale notaties xes en ys: zij $s = c + x + y$ dan $z = s \bmod 10$ en $c' = s \operatorname{div} 10$.

Voor elke positie van de uitkomst wordt het cijfer z als volgt bepaald. Zij s de som der cijfers op die

positie in de summanden, met daarbij ook nog de overdracht (carry) opgeteld. Dan is $z = s \underline{\text{mod}}\ 10 =$ het eenheden-cijfer van s, en de carry voor de volgende positie (één tienmacht verder) is $s \underline{\text{div}}\ 10$. De eerste carry, c_0, dient op nul gesteld te worden. Desgewenst kunnen de cijferrijen met insignificante nullen (tot gelijke lengte) aangevuld worden.

We geven nu bovenstaande algoritme op vier manieren weer; tweemaal zonder expliciete recursie en tweemaal met expliciete recursie. Daarna veralgemenen we de optelling van twee getallen tot een optelling van een stel getallen. We gaan er vanuit dat de optelling van cijfers (getallen in 0..9) met behulp van + gedaan kan worden. We gebruiken voorts de volgende globale definitie:

```
zeros :- oneindige lijst met cijfers nul
      = iterate id 0
```

Formulering 1 (zonder expliciete recursie)

We weten dat de uitkomst van de eenheden af gevormd wordt, d.w.z. te beginnen bij de positie der eenheden. (Dit staat niet uitdrukkelijk in mijn infor-

mele beschrijving van de algoritme, maar het is er wel uit af te leiden. We weten het ook al vanuit onze lagere school-ervaring.) Laten we de bewerking-per-positie $\oplus$ noemen. We merken nu op dat het resultaat van $\oplus$ in feite een tweetal is, nl. $(c', z : zs)$ waarbij z het nieuw gevormde cijfer van de uitkomst is, c' de nieuwe overdracht en zs de al bestaande cijferrij van de uitkomst (naar de eenheden toe; dus $zs ++ [z]$ bij de vlnr-representatie!). De algoritme bestaat dan in wezen uit de berekening van

$$\cdots (x,y) \oplus (\cdots (x_1, y_1) \oplus ((x_0, y_0) \oplus \text{basis}) \ldots) \cdots$$

Enerzijds heeft $\oplus$ de cijfers x en y als argument, en anderzijds het resultaat (c, zs) van de eraan voorafgaande bewerking $\oplus$. De definitie van $\oplus$ luidt dus als volgt.

$$(x, y) \oplus (c, zs) = (s \ \underline{\text{div}}\ 10, (s \ \underline{\text{mod}}\ 10) : zs) \quad \underline{\text{where}} \quad s = x + y + c$$

De optelling zelf definiëren we door:

```
xs plus ys = (finish . foldr (⊕) (0,[]) . zip) (xs', ys')
             where  xs' = take (m - #xs) zeros ++ xs
                    ys' = take (m - #ys) zeros ++ ys
                    m   = #xs max #ys
```

Hierbij zijn xs' en ys' van gelijke lengte, en ontstaan uit xs en ys door aanvulling met insignificante nullen. De functie finish zet de laatste overdracht gewoon op kop van de al verkregen cijferrij; we weten immers dat als alle cijfers in 0..g liggen, ook alle overdrachten in 0..g liggen en dus cijfers zijn.

finish (c, zs) = c : zs

Het is even eenvoudig om plus te definiëren in de veronderstelling dat de argumenten vlnr-representaties van getallen zijn (met de eenheden op kop). Dit laten we over aan de ijverige lezer.

Formulering 2 (zonder expliciete recursie)

Dit is een kleine variatie op bovenstaande beschrijving. Wellicht ligt het meer voor de hand om de bewerking-per-positie, ⊕, het tweetal (c', z) te laten opleveren (waarbij wederom c' de nieuwe overdracht is en z het nieuw gevormde cijfer van de uitkomst). De basis in

$$\cdots (x, y) \oplus (\cdots (x_1, y_1) \oplus ((x_0, y_0) \oplus \text{basis}) \cdots) \cdots$$

is nu het tweetal (0, dontcare) waarbij dontcare

er helemaal niet toe doet (en zelfs ⊥ mag zijn). Een bewerking ⊕ levert in feite niet louter het tweetal (c', z) op, maar zet deze op kop van de al gevormde lijst van dergelijke tweetallen. Na afloop van de ⊕ – bewerkingen wordt uit die lijst van tweetallen dan de gewenste cijferlijst gevormd. Aldus vinden we:

$$(x, y) \oplus ((c, z'') : czs) = (s \underline{\text{div}}\ 10,\ s \underline{\text{mod}}\ 10) : (c, z'') : czs$$
$$\underline{\text{where}}\quad s = c + x + y$$

$$xs \underline{\text{plus}}\ ys = (\text{map snd} \cdot \text{init} \cdot \text{foldr}\ (\oplus)\ [(0, \bot)] \cdot \text{zip})\ (xs', ys')$$
$$\underline{\text{where}}$$
$$(xs', ys') :- \text{als voorheen}$$
$$\bot :- \text{een of ander "don't care" cijfer}$$

Wie deze formulering in eerste instantie gevonden zou hebben, zou al snel tot de conclusie moeten komen dat het bewaren van de overdracht c in de bewerking $(x, y) \oplus ((c, z'') : czs)$ overbodig is: die overdracht wordt er verderop door (map snd) toch weer uit weggewerkt. De verandering van bovenstaande ⊕ in de vorige ⊕ (uit formulering 1) ligt dan voor de hand. Tevens kan dan init weggewerkt worden: init (_____ : [⊥]) = []. Een formeel bewijs dat deze en de vorige plus gelijkwaardig zijn, zal deze redenering op de voet volgen (en is niet moeilijk, denk ik).

Formulering 3 (met recursie)

Allereerst is er de mogelijkheid de manuele uitvoering van het lagere school-algoritme stap-voor-stap in de formele beschrijving te simuleren. Het resulterende "functionele" programma is dan in feite "procedureel" van aard. De functie plus hieronder krijgt steeds kleinere beginstukken van de oorspronkelijke xs en ys als argument, tesamen met de overdracht c van de voorgaande handelingen (die initieel 0 is).

```
xs plus ys = plus xs ys 0
plus (xs ++ [x]) (ys ++ [y]) c = plus xs ys c' ++ [z]
                                 where
                                 (c', z) = (s div 10, s mod 10)
                                 s = x+y+c
plus (xs ++ [x]) [] c = ...
plus [] (ys ++ [y]) c = ...
plus [] [] c = ...
```

Als we in de initiële aanroep van plus niet xs en ys meegeven, maar xs' en ys' (dus met voldoende nullen op kop aangevuld), dan kunnen de drie laatste clauses voor plus door één eenvoudige worden vervangen!

Bovenstaande vergelijkingen zijn géén legale definities,

(hoewel zij wel de plus eënduidig definiëren, en ook willekeurige lijsten maar op één manier als xs ++ [x] te ontleden zijn). Een legale definitie is hier echter gauw uit geformd:

```
xs plus ys = reverse (plus' (reverse xs) (reverse ys) 0)
plus' (x:xs) (y:ys) c = [z]: plus' xs ys c'
                        where
                        (c', z):- als voorheen
```

Het ligt nu voor de hand om in de representaties de eenheden op hop te zetten: de drie reverse-toepassingen in de definitie van xs plus ys kunnen dan achterwege blijven.

Formulering 4 (met recursie)

We geven nu een formulering (mét recursie) waarbij we niet proberen de manuele uitvoering van de algoritme stap-voor-stap te volgen. Integendeel, we proberen de essentie van de lagere school-algoritme te beschrijven door de functionele afhankelijkheid tussen uit- en invoer te formuleren.

Zij zs de gewenste uithomst van xs plus ys. Dan vinden we achtereenvolgens de volgende "definities" voor zs (en enige andere hulpgrootheden).

```
zs   :-  cijfers van de uithomst
     =   map (mod 10) ss
ss   :-  som der cijfers-en-carry per positie
     =   map sum xycs
xycs :-  cijfers-en-carry per positie
     =   zip (zip (xs', ys'), cs)
cs   :-  overdrachten per positie
     =   tail (map (div 10) ss ++ [0])
```

(Alleen in de laatste regel is er gebruik gemaakt van de unit-eigenschap van de representatie.) Deze vergelijkingen vormen, als definitie beschouwd, een circulaire definitie, omdat ss en cs in elkaar zijn uitgedrukt. Dat dit geen principieel bezwaar is, voor dit algoritme, weten we al van de lagere school: er is een berekeningsmethode die niet alsmaar in een kringetje doordraait. Immers, uit de definities blijkt dat de laatste van cs well-bepaald is, nml. 0, en dus ook de laatste van xycs en van ss. En als de laatste van ss bepaald is, dan ligt daarmee ook de een-na-laatste van cs

vast, en dus ook de een-na-laatste van xeycs en van ss. Enzovoorts!

Helaas blijft Lazy Evaluation (en iedere andere Herschrijf-Evaluatie) wél in een keringetje doordraaien zonder productief te zijn. We kunnen dit op twee manieren beredeneren. De eerste, "operationele", manier is als volgt. Bij gebruik van de definities als herschrijf-regels (linkerlid door rechterlid vervangen, in willekeurige expressie), zal blijken dat zowel ss, alsook xeycs en cs niet bepaald kúnnen worden. (Probeer het maar eens met xes' = ys' = [] !) De tweede, "wiskundiger" manier is als volgt. Het is eenvoudig na te rekenen dat de waarden zs = ss = xeycs = cs = ⊥ óók een oplossing van het stelsel vergelijkingen vormt,[F] en dus de "minst informatieve" oplossing is. ((Elke waarde "is minstens zo informatief als" ⊥, en de relatie "is minstens zo informatief als" is monotoon in de componenten van samengestelde waarden.)) Het is een feit dat 'n Herschrijf-Evaluatie de minst informatieve oplossingen (bij recursie-vergelijkingen) oplevert!

[F] want cs = tail(⊥ ++ [0]) = tail(⊥) = ⊥,

We krijgen een productief stel recursieve definities als we de vlnr-representatie kiezen (met de eenheden op kop):

```
zs, ss, xycs :- als tevoren
cs = init ( [0] ++ map (div 10) ss)
```

Ook al zou ss = ⊥, dan toch is cs = init ([0] ++ map (div 10) ⊥) = init ([0] ++ ⊥) = init (0:⊥) = 0:⊥ ≠ ⊥. Er geldt dus zeker (head cs) = 0 zodat ook zeker (head ss) wel-bepaald is (≠ ⊥) en daarmee ook het tweede element van cs, enzovoorts.

Voor de vlnr-representatie luidt de definitie dus:

```
xs plus ys = map (mod 10) ss
             where
             ss = map sum xycs
             xycs = zip (zip (xs', ys'), cs)
             cs = init ([0] ++ map (div 10) ss)
             sum ((x,y), c) = x+y+c
             xs' = take m (xs ++ zeros)
             ys' = take m (ys ++ zeros)
             m = #xs max #ys
```

Optelling van een stel getallen

Om lijsten [xs, ys, ...] van getallen op te tellen kunnen we de operatie plus veralgemenen tot lijsten. Eén manier om dat te doen, is als volgt:

```
plus = fold (plus) [0]
```

(met foldl of foldr voor fold)

met foldr of foldl voor fold. Maar op grond van onze lagere school-ervaring weten we dat het ook iets efficiënter kan: het gehannes met de overdrachten kun je tot éénmaal per positie beperken door per positie de cijfers van álle summanden op te tellen. Deze veralgemening is eenvoudig in de gegeven definities van plus door te voeren. We doen dat ter illustratie hier voor Formulering 1.

Hier is, ter herinnering, de definitie van plus.

```
xs plus ys = (finish • foldr (⊕) (0,[]) • zip) (xs', ys')
                where xs', ys' :- ...
(x,y) ⊕ (c,zs) = (s div 10, (s mod 10) : zs)  where s = x+y+c
finish (c,zs) = c : zs
```

De veralgemening tot lijsten luidt als volgt; (een toelichting volgt daarna nog).

```
plus xss = (finish' • foldr (⊕') (0,[]) • zipl) xss'
           where
             xss' = [take (m - #xs) zeros ++ xs | xs ← xss]
             m = max [#xs | xs ← xss]
xs ⊕' (c, zs) = (s div 10, (s mod 10):zs) where s = sum xs + c
finish' (c,zs) = repr c ++ zs
repr = reverse • map (mod 10) • takewhile (≠0) • iterate (div 10)
```

De veralgemeningen van xs', ys' tot xss', van zip tot zipl (zie onder), van ⊕ tot ⊕' en van finish tot finish' spreken voor zich. De functie repr levert de unit-representatie van een getal. Anders dan bij de optelling van twee getallen is er nu geen vaste bovengrens (zoals 9) aan te geven voor de grootte van de overdracht. Vandaar dat de heel algemeen toepasbare functie repr gebruikt wordt.

Er rest ons nog zipl te definiëren. Er moet gelden

$$\text{zipl}\ [[x_1, x_2, \cdots], [y_1, y_2, \cdots], \cdots] = [[x_1, y_1, \cdots], [x_2, y_2, \cdots], \cdots]$$

Dus de kop van (zipl xss) is (map head (xss)), en daarna komt (map head (map tail xss)), en dan (map head ((map tail • map tail) xss)) enzovoort. De definitie luidt derhalve als volgt.

```
zipl = map (map head) • takewhile allNE • iterate (map tail)
       where allNE = all • map (≠[])
```

((Alweer is hier het idioom (map... • takewhile ... • iterate ...) te zien, net zo als in de definitie van repr. Op den duur gaan deze idiomen als een conceptuele eenheid gezien worden, net zo als ax^2+bx+c in de rekenkunde. En evenmin als er in de rekenkunde voor elk idioom een apart symbool wordt geïntroduceerd, hoeft er voor het map-takewhile-iterate idioom een aparte functie-constante geïntroduceerd te worden.))

Het was wederom een verrassing voor mij dat zipl zo helder, zónder expliciete recursie gedefinieerd kan worden. Voordat ik mijn pogingen deed om recursie-loos te programmeren, was hetvolgende voor mij "de natuurlijke" definitie.

```
zipl [xs]     = map mklist xs
zipl (xs: xss) = (map cons • zip) (xs, zipl xss),  if xss ≠ []
mklist x = [x]
cons (x, xs) = x: xs
```

(Er geldt ook mklist = (:[]) zodat een aparte definitie van mklist niet eens nodig is.)

Het vermenigvuldigen van getallen

Na de uitgebreide behandeling van de optelling valt hier niet veel bijzonders meer te melden. Ik zal daarom heel kort zijn in mijn uitleg, en alleen het analoge van Formulering 1 geven, (dus zonder expliciete recursie, en met de eenheden aan de staart van de lijst).

De informeel gegeven algoritme staat schematisch in de volgende figuur.

$$
\begin{array}{rl}
x_m \text{ ----- } x_1\ x_0 & = \text{xs} \\
y_n \text{ -- } y \text{ -- } y_0 & \\
\hline
\times & \\
z^0_{m_0} \text{ ---- } z^0_1\ z^0_0 & = y_0 \text{ times xs } +\!+ [\,] \\
\vdots & \\
z_{m'} \text{ ----- } z_1\ z_0\ 0..0 & = y \text{ times xs } +\!+ [0,\ldots,0] \\
\vdots & \\
z^n_{m_n} \text{ -----} z^n_1\ z^n_0\ 0 \text{ ---- } 0 & = y_n \text{ times xs } +\!+ [0,\ldots,0] \\
\hline
+ & \\
\text{de uitkomst xs } \underline{\text{mult}} \text{ ys} &
\end{array}
$$

We definiëren xs mult ys met behulp van een operatie y times xs die een cijfer met een getal vermenigvuldigt. De definities spreken voor zich:

```
y times xs = (finish • foldr (⊗) (0,[])) xs
             where
               x ⊗ (c, zs) = (s div 10, (s mod 10) : zs)
                             where s = c + y × x
               finish (c, zs) = c : zs
```

```
xs mult ys
  = plus [ys!i times xs ++ take (i-1) zeros | i ← [1..#ys]]
  = (plus • map app • zip) (map (times xs) ys, iterate (0:) [])
    where app (xs, ys) = xs ++ ys
```

De tweede, alternatieve, definitie voor mult maakt geen gebruik van indices en indicering, en is daarom in zekere zin eleganter dan de eerste definitie. Die eerste definitie is overigens wel iets duidelijker, vind ik.

* * *

De volgende oefening is natuurlijk om de staartdelingsalgoritme van de lagere school in Bird's notatie -- zonder expliciete recursie -- uit te drukken. Ik wens u veel plezier.

—4—