Magische woordvierkanten

Maarten Fokkinga
Vakgroep SETI

Versie van 12 november 1995

Probleemstelling Gegeven is een lijst van woorden. Gevraagd wordt alle mogelijke 4×4-vierkanten van letters waarin elke rij en elke kolom een woord is uit de gegeven woordenlijst *wrds*. Het getal 4 is hierbij natuurlijk een parameter:

```
size = 4
wrds = ...
```

Rieks op den Akker heeft me dit probleem gesuggereerd. Hij maakte me erop attent dat in de oplossing een leuke toepassing zit van bomen met een variabele vertakkingsgraad: *tree* ::= *Fork char* [*tree*]. Bovendien blijkt er in ons programma nog een leuke datastructuur voor de hand te liggen: een vierkant van bomen, [[*tree*]]. Het is ondoenlijk om, zonder verregaande abstractie, een plaatje van zo'n vierkant te tekenen (maar Miranda heeft er geen problemen mee: bijna alle data worden door verwijzingen, pointers, gerepresenteerd).

Analyse Het probleem valt typisch in de test-and-generate klasse: genereer alle vierkanten en schrap daarvan al diegene waarvan niet alle rijen en kolommen in de woordenlijst voorkomen. Maar da's niet efficiënt. Beter is het om *tijdens* het generatie-proces bij *iedere* keuze voor de invulling van een plaats in het vierkant te testen of die keuze misschien al foutief is (niet meer tot een woord in de woordenlijst is uit te breiden). Anders gezegd, voor iedere invulling van een plaats in het vierkant willen we uitsluitend letters kiezen die de alreeds gevormde woord-stukken uitbreiden tot woord-stukken die in de woordenlijst voorkomen.

Daarvoor hebben we een datastructuur nodig die bij ieder woord-stuk alle toegestane uitbreidingen opsomt. Zo'n datastructuur is een *boom* met één karakter per knoop en een variabele vertakkingsgraad: de karakters op een pad vormen samen een woord, en iedere subboom geeft de mogelijke uitbreidingen aan van het woord-stuk dat staat op het pad vanaf de wortel tot aan die subboom. Om het een en ander tijdens de ontwikkelingsfase te testen is het wel handig om direct ook geschikte show-functies te definiëren:

```
tree  ::= Fork char [tree]
showT = showT' 0
showT' n (Fork c ts)
= spaces n ++ [c] ++ "\n", if ts=[]
= spaces n ++ [c]++ drop (n+1) (showTs' (n+1) ts), otherwise
showTs = showTs' 0
showTs' n = concat . map (showT' n)
```

(Let op de slimme truc met '*drop*$(n{+}1)$', waardoor niet alle karakters op een aparte regel komen te staan.) Om de onderdelen van een boom aan te duiden, op plaatsen waar pattern matching te veel schrijfwerk vereist, definiëren we nog twee functies:

```
sucs (Fork c ts) = ts
top  (Fork c ts) = c
```

De gegeven woordenlijst willen we opslaan in een lijst van bomen. Dat gebeurt woord-voor-woord door de functie *mktrees*:

```
mktrees :: [[char]] -> [tree]
mktrees = foldr into []
```

We zullen *into* zó definiëren dat de uitkomst van *into xs ts* onstaat uit de bomenlijst *ts* door één van de bomen uit te breiden met de gegevens van woord *xs*. De op te leveren bomenlijst, en ook iedere bomenlijst 'onder een *Fork*', is ge-ordend; 'van links naar rechts' met opklimmende waarde van het karakter in de top. Van die eigenschap wordt in de definitie gebruik gemaakt:

```
into :: [char] -> [tree] -> [tree]
into [] ts = ts
into (x:xs) ts
 = ts0 ++ [Fork x (into xs [ ])] ++ ts1, if ts1=[] \/ x~=c
 = ts0 ++ [Fork x (into xs ts')] ++ ts2, otherwise || dus hier: x=c
where
ts0 = takewhile ((x<).top) ts
ts1 = dropwhile ((x<).top) ts
(Fork c ts'): ts2 = ts1
```

We representeren nu de relevante woorden van de gegeven lijst:

```
dictionary = mktrees (filter p wrds)     where   p = (size=) . (#)
```

Het blijkt achteraf, en bij nader inzien zouden we dat zelf ook wel *nu* kunnen bedenken, dat met deze definitie de woorden onderste-boven en achterste-voren staan. Dat is eenvoudig te verhelpen met een transformatie van het vierkant vlak voor het afdrukken, of door een ietwat gewijzigde definitie van *dictionary*:

```
dictionary
= mktrees (map reverse (filter p wrds))     where  p = (size=) . (#)
```

Het genereren We representeren een vierkant, of algemener: een $k{\times}n$-rechthoek, door een n-lijst van k-lijsten; we noemen de k-lijsten *rijen*. Op elke plaats in een rij staat niet zozeer één letter, als wel een *boom*, waarvan de top de bedoelde letter is, en de subbomen de mogelijke uitbreidingen aangeven van het woord. Bij het tonen van een rechthoek moet van iedere boom dus eerst de *top* worden genomen:

```
matrix == [[tree]]
showM  :: matrix -> [char]
showMs :: [matrix] -> [char]
showM   = lay . map (map top)
showMs  = lay . map showM
```

Het generatie-proces van de $k \times n$-rechthoeken volgt het standaard patroon:

```
gen k 1 = alle_mogelijke_eerste_rijen_ter_lengte  k
gen k (n+2)
= [rij: m | m <- gen k (n+1); rij <- volgenden_bij m; rij $past_bij m]
```

Laten we zeggen dat de rechthoek *van beneden naar boven* gegenereerd wordt: in '*rij* : *m*' staat *rij* bovenop *m*. Op iedere plaats in de rechthoek staat de boom die de toegestane uitbreidingen *naar boven toe* opsomt. De vorming van de mogelijke nieuwe rijen, bovenop een gegeven rij *xs*, gebeurt door *nexts xs*. De hulpfunctie *nexts′* is in essentie dezelfde als *nexts* maar levert tevens de boom *u* op die de mogelijke uitbreidingen-*naar-links* opsomt:

```
nexts :: [tree] -> [[tree]]
nexts (x:xs)
= [y:ys | (u, ys) <- nexts' xs; y <- sucs x; top u = top y]

nexts' :: [tree] -> [(tree,[tree])]
nexts' [] = [ (u,[]) | u <- dictionary]
nexts' (x:xs)
= [(v, y:ys) | (u, ys) <- nexts' xs; y <- sucs x; top u = top y; v <- sucs u]
```

Merk op dat, inderdaad, de nieuwe *y* komt uit *sucs x*; de top ervan moet wel kloppen met wat er links van *ys* mag staan (gerepresenteerd door *u*). Merk ook op dat, inderdaad, de uitbreiding naar links (gerepresenteerd door *v*) niet bepaald wordt door *y* maar door *u*.

Omdat iedere boom ge-ordend is, kan het deel '*ysucs x*' vervangen worden door:

```
y <- (take 1 . dropWhile ((top u <) . top) . sucs) x
```

Maar dat levert bij kleine vertakkingsgraden geen efficiëntiewinst op; de overhead van de extra functie-aanroepen is te groot.

Voor de generatie van alle mogelijke eerste rijen kunnen we, toevallig, ook functie *nexts* gebruiken. Immers, een eerste rij is louter een uitbreiding bovenop een 'nulde rij' die uit *k* 'superbomen' bestaat (waarvan de top steeds niet terzake doet):

```
firsts :: num -> [[tree]]
firsts k = nexts (rep k super)      where  super = Fork undef dictionary
```

Na al deze voorbereidingen zijn we nu in staat het generatie-proces *gen* te definiëren volgens de bovengegeven opzet. De conditie '*rij* $*past_bij m*' verdwijnt echter omdat, per constructie, iedere volgende rij *xs* (geproduceerd door *next ys*) past bij de rij *ys* eronder. Dus:

```
gen :: num -> num -> [matrix]
gen k 1 = [[xs] | xs <- firsts k]
gen k (n+2) = [xs:ys:m | ys:m <- gen k (n+1); xs <- nexts ys]
```

De gevraagde lijst van alle vierkanten wordt getoond door *test*:

```
test = showMs (gen size size)
```