Gesynchroniseerde interactie onder lazy evaluation

Maarten Fokkinga, 3 juli 1985.

werkexemplaar

Bij een dialoog tussen computer en gebruiker, die in een lazy-geevalueerde taal geprogrammeerd is, doet zich het verschijnsel voor dat soms al delen van een antwoord op het beeldscherm verschijnen al vordat de gegevens door de gebruiker zijn ingetikt. Wij geven hieronder aan hoe zulks vermeden kan worden.

* * *

Als doorlopend voorbeeld beschouwen we de programmering van de volgende dialoog.

(1) Op het beeldscherm verschijnt de vraag "First?"
(2) Op deze vraag tikt de gebruiker een getal in, zeg y.
(3) Op het beeldscherm verschijnt:
"squared:" gevolgd door y^2
en dan op een nieuwe regel de vraag "Next?"
(4) goto (2).

De eerste poging om dit te programmeren luidt als volgt. (We gebruiken een SASL-achtige notatie.) Definieer


```
q X = "First?": p X
p X = "squared:": y*y: nl: "Next?": p Y
      where y, Y = hd X, tl X
```

De dialoog wordt dan in werking gesteld door het commando

q interactive

Helaas voldoet dit programma niet helemaal aan de specificatie: een deel van wat volgens (3) op het beeldscherm moet verschijnen, staat daar al vóórdat de gebruiker het onder (2) gevraagde getal heeft ingetikt. Immers, er geldt (volgens lazy evaluation)

```
q X = "First?": p X
    = "First?": ("squared:": ....  )
    = "First?": "squared:": (....  )
```

Inderdaad, de door p X op te leveren lijst heeft een kop die geheel onafhankelijk is van X en dus al geproduceerd wordt zonder dat X (en met name z'n kop) hoeft te bestaan. Wij zullen hieronder vijf technieken schetsen waarmee dit ongewenste effect voorkomen kan worden; ze zijn gerangschikt volgens toenemende voorkeur.

Opmerking Het is ook mogelijk dat de gebruiker zich niet aan de specificatie houdt, door nml een volgend getal alvast in te tikken nog voordat de vraag daarom op het beeldscherm is verschenen. We zullen ons hierom niet bekommeren. In ons voorbeeld is zo'n mogelijkheid zelfs gebruiksvriendelijk te noemen. Desgewenst is er nog wel iets tegen te doen door vanuit het programma het toetsenbord te blokkeren; daarbij zal waarschijnlijk ook van onderstaande technieken gebruik gemaakt moeten worden. (Einde opmerking)

Methode 1: bedrog

We laten het ingetikte getal direkt op het beelscherm verschijnen, vóór enig ander deel van de respons. Maar hiermee plegen we wel bedrog, want het echo-en van de invoer is niet in de programma-specificatie opgenomen. De nieuwe definitie voor p luidt

```
p X = y: "squared:": y*y: nl. Next?": p Y
      where y, Y = hd X, tl X
```


Methode 2: taaluitbreiding

We breiden de taal uit met zgn. value parameters. Semantisch is het effect van een value parameter dat bij aanroep het argument wordt uitgerekend (geevalueerd) alvorens met de evaluatie van de functieromp wordt begonnen. Met value parameters kan dus de lazyness van de evaluatie lokaal doorbroken worden. Value parameters zijn zo-wie-zo al nodig om ongewenst groot geheugenbeslag te voorkomen dat soms t.gv lazy evaluation optreedt. Het programma luidt nu als volgt.

```
q X = "First?": p (hd X) (tl X)
p (value y) Y = "squared:": y*y: nl: "Next?": p (hd Y) (tl Y)
```

Deze oplossing is weinig elegant, maar wel korrekt.

Methode 3: redundante tests

We maken op kunstmatige, ja zelfs enigszins gekunstelde, manier het deel "squared:" afhankelijk van y door het binnen een then- of else-tak van een test op y te zetten. Bijvoorbeeld als volgt.

```
p X = (y=0 → rest; rest)
        where  rest = "squared:": y*y : ---
               y, Y = hd X, tl X
```

of

```
p X = (y=0 → ""; ""): "squared:": y* y; ---
        where  y, Y = hd X, tl X
```

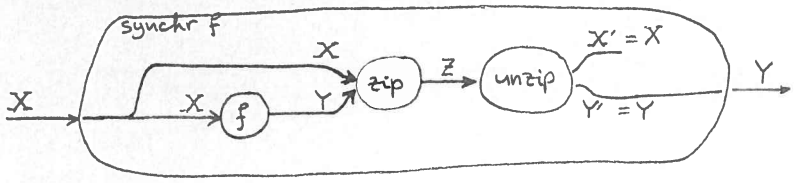
Dit is nog wel de minst elegante oplossing; maar de voorgaande oplossingen waren beide enigszins bedrog.

Methode 4: geprogrammeerde synchronisatie

We definieren een funktie synchr zó dat (synchr f) = f voor lijsttransformaties f; het operationele effect is echter dat ieder element uit de resultaatlijst van f een beer gehoppeld is geweest aan het overeenkomstige element in de invoerlijst van f. Die hoppeling en ontkoppeling gebeurt door zip en unzip: zip maakt van een tweetal lijsten een lijst van tweetallen en unzip doet het omgekeerde (zip = ritssluiting).

```
zip (x:X) (y:Y)  = [x, y]: zip X Y
unzip ([x, y]: Z) = [x:X, y:Y] where [X, Y] = unzip Z
synchr f X = Y'
        where [X', Y'] = unzip (zip X Y)
              Y = f X
```


Een grafische voorstelling van (synchr f) ziet er als volgt uit.



We wijzigen de definitie van p nu zo dat de resultaatlijst elementsgewijze past bij de invoerlijst, en onderwerpen die funktie dan aan synchr. Het programma luidt dan

```
q X = "First?": synchr p X
p X = ["squared:", y*y, nl, 'Next?"]: p Y
        where y, Y = hd X, tl X
```

Dit is een redelijk elegante oplossing en lijkt redelijk algemeen toepasbaar. Merk op dat we de ~~oorsp~~ funktie p wel iets hebben moeten wijzigen. Maar dat is niet zo erg, want dat zou toch gemoeten hebben als we het programma zouden willen typeren: nu is het type van p

nbrlist → [charlist, nbr, char, charlist] list

waarbij [t, t', ..., t''] een zgn. record-type is. Alle vorige versies zouden niet zo eenvoudig (of zelfs helemaal niet) te typeren zijn! [Recordwaarden zijn lijsten van statisch bepaalde lengte, alleen te noteren met de expliciete opsommingnotatie [w, w', ..., w''] en alleen te ontleden middels zgn. patternmatching constructies in formele parameterpatterns en case-patterns.
Voor een voorbeeld, zie zip en unzip!]

Methode 5: robuuste programmering

We geven eerst de oplossing en lichten daarna de titel van deze methode toe. De oplossing bestaat hierin dat de funktie p eerst test of y wel bestaat alvorens "squared:": y*y: ... op te leveren. Herinner je dat volgens methode 3 getest werd op de waarde van y; nu dus slechts op het bestaan van y ofwel er wordt getest of X de vorm (y:Y) heeft. De definitie van p komt dan te luiden:

en al X=[]!

```
p (y:Y) = "squared:": y*y: "nl": "Next?": p Y
```

Deze definitie is zo natuurlijk (mede gezien mijn gewoonte om zo mogelijk robuust te programmeren) dat

het enige tijd duurde voordat ik de definitie
p X = ... where y, Y = hd X, tl X had gevonden en daarmee een voorbeeldprogramma met het ongewenste gedrag!

Als formele parameterpatterns niet zijn toegestaan dan kunnen we wellicht een case-constructie gebruiken (zoals in TWENTEL) of zoiets zelf programmeren:

```
p X = X=[] → .... ;
      "squared:": y*y: nl: "Next?": p Y
      where y, Y = hd X, tl X
```

De hier gepresenteerde programma's zouden we ook gevonden hebben als we louter hadden gestreefd naar een robuuste programma. We noemen een programma of programmadeel robuust als onbedoelde invoer niet onopgemerkt blijft maar leidt tot een foutmelding of foutstop of desnoods tot nonterminatie-zonder-enig-resultaat; wat dus niet mag gebeuren is dat de evaluatie normaal eindigt met een resultaat dat ook als het resultaat van bedoelde, legale invoer is te interpreteren. Robuustheid is net als correctheid een eis die aan alle programmatuur gesteld moet worden; beide zijn noodzakelijk voor betrouwbaarheid van de resultaten.

Welnu, de test op het niet leeg zijn van X is louter een robuustheidsvoorziening, want uit de specificatie blijkt dat het niet de bedoeling is dat de invoer ooit ophoudt: p heeft (en mag!?) dus niet gedefinieerd zijn voor de lege lijst. Oh, zo

Wellicht ten overvloede merken we op dat de te vroege verschijning van "square:" zich zo-wie-zo niet zou voordoen als de invoer ook eindig zou mogen zijn. In dat geval namelijk is een test op het leeg zijn van X in p onvermijdelijk. De elegantste manier is natuurlijk weer via een pattern-match op de parameter:

```
p []    = "over en sluiten": []
p (y:Y) = 'squared:': y*y: nl: 'Next?': pY
```

* * *

Tot besluit

Het hele probleem komt voort uit het feit dat de programmaspecificatie zich niet alleen over de input-output-relatie uitspreekt, maar ook over tijdsaspecten. Tijdsaspecten worden niet betrokken in het conventionele begrip semantische equivalentie = . Alle bovenstaande

oplossingen zijn semantisch equivalent en dus ononderscheidbaar met betrekking tot = . De redeneringen die wij hebben gehouden zijn gebaseerd op kennis van de implementatie, en dus ook implementatie-afhankelijk. Het lijkt mij dat het problematische tijdsaspect net zo implementatie-afhankelijk is als kostenaspecten zoals tijdsduur en geheugenbeslag.