Van FP naar IP – een case study

Maarten Fokkinga, 7 april 1989.

We beschouwen Bird's priemgetallenprogramma en proberen dat op gedisciplineerde manier in een imperatief programma om te zetten. Het blijkt in dit geval dat kennis van het functionele priemgetallenprogramma de constructie van het Pascal-programma nauwelijks vergemakkelijkt.

* * *

Inleiding

In deze case study onderzoek ik in hoeverre gedegen ervaring in functioneel programmeren van nut is bij de constructie van imperatieve programma's. Met name ben ik geïnteresseerd in de vraag in hoeverre een gedegen beheersing van imperatief programmeren nodig is bij een gedisciplineerde omzetting van alreeds ontwikkelde functionele programma's naar imperatieve programma's. Het antwoord op deze vraag is van groot belang bij de inrichting van het onderwijs in het programmeren, wanneer o.a. als einddoel wordt gesteld dat niet

alleen functioneel programmeren beheerst moet worden, maar ook de omzetting van functionele programma's naar imperatieve.

De bevindingen in deze ene case study zijn als volgt. Wanneer het einddoel een imperatief programma is, dan is beheersing van functioneel programmeren --in onze case study-- in geen dele een reden om het imperatief programmeren minder te hoeven beheersen. Ogenschijnlijk lijkt het programmeren alleen maar "moeilijker" te worden door de twee-zijdigheid functioneel + imperatief. Dit is ook wel zo; maar als resultaat is de programmatuur wel van hogere kwaliteit -- althans er is meer gelegenheid om het programmeertraject met wishandige gestrengheid af te leggen.

Bovenstaande bevinding wordt ook bevestigd door de omzetting van functioneel geformuleerde backtracking programma's naar imperatieve programma's; zie [Fokkinga 1986, 1987a, 1987b]. Ook daarbij is een gedegen beheersing van de imperatieve uitdrukkingswijzen een noodzakelijke voorwaarde om de vruchten te plukken van de functionele formuleringen.

De case study

We gaan uit van het volgende programma:

```
primes = map head (iterate sieve [2..])
sieve(p:xs) = [x | x ← xs; x mod p ≠ 0]
```

Hierbij is iterate een standaardfunctie: iterate f x = [x, f x, f(f x), ...]. We willen nu een priemgetallen-programma in Pascal construeren. Het is weinig zinvol om lazy evaluation en oneindige lijsten exact te simuleren, want zo'n simulatie zal niet efficienter zijn dan de Miranda-implementatie zelf. We willen een "net" Pascal-programma waarvan de functionele herkomst niet "afdruipt".

Keuze 1. We willen met arrays werken in Pascal. Dus slechts een eindig deel van primes wordt door het Pascal-programma opgeleverd, zeg priemgetallen in het bereik 2..n. We leiden af (in de functionele notatie):

```
takewhile (≤n) primes
  = map head (takewhile (≠[]) (iterate sieve [2..n]))
  = (map head · takewhile (≠[]) · iterate sieve) [2..n]
```

(NB. In een eerste schets van dit verhaal was ik het deel (takewhile (≠[])) vergeten!)

Keuze 2. We kiezen één (globaal) array waarin we voor $k = 0, 1, 2, \ldots$ achtereenvolgens in de tijd de tussenresultaten $(sieve^k\ [2..n])$ opslaan. Dit is gerechtvaardigd, omdat $(sieve^k\ [2..n])$ alleen maar nodig is voor (i) de head ervan en (ii) de berekening van $(sieve^{k+1}\ [2..n])$. Het ziet er naar uit dat de berekening van sieve op een argument in situ kan gebeuren. We noemen het array

var present: array [2..n] of boolean

en de interpretatie hiervan is de lijst $[j \mid j \leftarrow [2..n];$ present$[j]]$.

Keuze 3. We introduceren een hulpvariabele p : integer die "de head van present" aanwijst. We nemen dus als (extra) representatie-invariant:

$$\text{all}\ [\sim \text{present}[j] \mid j \leftarrow [1..p-1]] = \text{True}$$
$$\wedge\ (p = n+1 \vee \text{present}[p] = \text{True})$$

(NB. De symbolen $=, \wedge, \vee$ hebben hier hun wiskundige betekenis. Desgewenst kun je ze ook als

Miranda-symbolen opvatten, maar dan is het geheel louter één grote Miranda-expressie van type bool en luidt de invariant dat die expressie wishandig gelijk is aan True. Deze dubbelzinnigheid heeft verder geen invloed.)

Implementatie 1. De berekening van $sieve^{k+1}\ [2..n]$ uit $(sieve^{k}\ [2..n])$ luidt nu --in termen van de gekozen representatie-- als volgt:

"sieve":

```
var j: integer;
j:=p; present[p]:= false;
while j+p ≤ n do
begin j:=j+p; present[j]:= false end;
{nu: herstel van (extra) representatie-invariant}
while p≤n and not present[p] do p:=p+1
```

Implementatie 2. De berekening van ((map head. takewhile (≠[]) . iterate sieve) [2..n]) luidt nu als volgt:

"iterate":

```
"initialize";
while "list non-empty" do
begin "take head"; "sieve" end
```

waarbij -- op grond van de representatiekeuzen 2 en 3 --

"initialize": for j:=2 to n do present[j] := True; p:=2

"list non-empty": p ≤ n

"take head": write (output, p)

Correctheidsoverweging. Ook al is het functionele programma per aanname (of per definitie) correct, dan toch is er enig bewijs nodig voor de correctheid van de imperatieve implementatie ten aanzien van het gegeven uitgangspunt. Deze correctheidsoverwegingen kunnen in de vorm van invarianten geformuleerd worden. We beschouwen alleen de eerste repetitie van "sieve" en de hoofd-repetitie van "iterate". De repetities aan het eind van "sieve" en die in "initialize" zijn minder interessant.

Allereerst de repetitie van "sieve". Bij 'keuze 2' is de interpretatie van array present al vastgelegd. We breiden die interpretatie nu uit tot segmenten van present, als volgt:

$$\mathcal{I}\,(\text{present}[i..j]) = [x \mid x \leftarrow [i..j];\ \text{present}[x]]$$

De/een invariant van de repetitie luidt nu:

Zij $present_0$ de waarde van present aan het begin van "sieve". Dan geldt direct na 'while':

$$sieve\ J(present_0) = J(present[2..j+p-1]) \mathbin{++} [x \mid x \leftarrow J(present[j+p..n]);\ x \bmod p = 0]$$

$\wedge$ j is een veelvoud van p

De verificatie van de invariantie laat ik hier achterwege; die is weinig interessant. Wel merk ik op dat na afloop van de repetitie geldt $j+p \notin n$ en dus, tesamen met de invariant, volgt:

$$sieve\ J(present_0) = J(present)$$

Daarmee is de correctheid van dit deel aangetoond. Overigens zie ik, nu dat ik dit schrijf, dat een getrouwere vertaling van het functionele programma als volgt luidt:

```
"sieve":
  var j: integer;
  j:=p;
  while j≤n do
  begin if present[j] then
          if j mod p=0 then present[j]:=false;
    j:=j+1
  end .....
```

Hadden we de omzetting naar Pascal helemaal binnen het functionele programma willen voorbereiden, dan hadden we eerst een equivalent programma moeten afleiden dat met boolean lijsten werkt. In principe is dit wel mogelijk en is dit zelfs de juiste methode; in de praktijk is zo een nauwgezetheid ondoenlijk.

Nu de repetitie van "iterate". De/een invariant luidt hier:

```
takewhile (≤n) primes =
    output ++ (map head . takewhile ≠[] . iterate sieve) J(present)
```

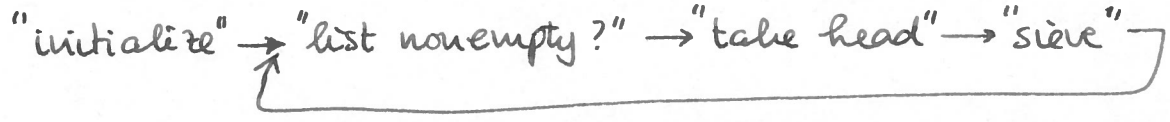
De verificatie van de invariantie berust op de eigenschap dat, voor J(present) ≠ []:

```
(map head . takewhile (≠[]) . iterate sieve) J(present)
= [head J(present)] ++ (map head . takewhile (≠[]) .
                        iterate sieve) (sieve J(present))
```

Uit de invariant volgt met name dat na afloop van de while geldt:

```
takewhile (≤n) primes = output
```

en daarmee is de correctheid van dat deel aangetoond. Merk overigens ook op dat er een exacte overeenkomst is tussen



en de berekeningsstappen t.g.v. "[2..n]", "iterate sieve", "takewhile (≠[])" en "map head" in het functionele programma.

Tenslotte nog een woord over "de representatie-invariant" uit kenzepunt 3. Die invariant is niet gekoppeld aan één specifieke repetitie, maar hij is gekoppeld aan de in- en uitgangen van de implementaties van "sieve", "take head" etc. Binnen die implementaties is de representatie-invariant misschien tijdelijk verstoord. Maar erbuiten, dus aan het het begin van die implementaties, is hij steeds waar. In feite is variabele p redundant; maar de waarheid van de representatie-invariant stelt "sieve" en "list non-empty" en "take head" in staat om hun berekening efficiënter uit te voeren. Het is dan ook ieders plicht om die representatie-invariant in stand te houden.

Terugblik

Ik kan mij niet voorstellen dat bovenstaand verhaal zou passen (of zelfs maar begrepen zou kunnen worden) in een TELEAC-cursus Pascal. Toch kan ik mij wel voorstellen dat de cursisten van de TELEAC cursus in staat zijn een priemgetallenprogramma te schrijven. De vraag komt derhalve op of al dat gedoe met functionele programma's, de vertaling ervan naar Pascal en representatie-invarianten etc. nog enig nut heeft, of alleen maar een ballast is.

Ik denk dat de weg via functionele programma's inderdaad een extra inspanning kost. Maar niet zonder nut. De winst die je ermee behaalt is dat het programmeertraject nu voor een groter deel vatbaar is voor bewijsvoering en dat daardoor de geleverde programmatuur van hogere kwaliteit kan zijn. Wanneer de kwaliteit van programmatuur slechts een (heel) ondergeschikte rol speelt, dan is "al dat gedoe" ook zeker niet nodig en kun je beter niet nadenken en direct maar gaan doen.

Verwijzingen

Fokkinga, M.M., Backtracking and Branch-and-Bound functionally expressed. Memorandum INF-86-18, june 1986, University of Twente.

Fokkinga, M.M., Transformatie van Specificatie tot Implementatie. In: colloquium Software Specificatie Technieken, eds. J.G. Schoenmakers, J.J.A.M. Poirters. Te verschijnen in 1987

Fokkinga, M.M., Exercises in Transformational Programming – Backtracking and Branch-and-Bound. In preparation - to be finished before May 1987.