Afscheid van Jaco

Aan prof.dr. Jaco de Bakker
ter gelegenheid van zijn afscheid van het CWI.

Beste Jaco,

Als je dit leest zit ik op een conferentie in Hongarije, dus deze brief is tevens mijn persoonlijke afscheid van jou. De brief bestaat uit twee delen: een wetenschappelijk deel en een kort persoonlijk naschrift.

De Thue Morse reeks

Mijn wetenschappelijke bijdrage sluit aan bij het stuk van Jan Willem Klop in deze zelfde afscheidsbundel, dat ik van Jan Willem onder embargo te lezen heb gekregen. Je zult je herinneren dat Jan Willem in de CWI lezing ter gelegenheid van zijn eredoctoraat kort refereerde aan de Thue Morse reeks. Noem deze reeks M. Jan Willem gaf de versie die start met 1. Noem het resultaat van omwisselen van nullen en enen in de Thue Morse reeks M'. De reeks M' is wat je krijgt als je het Thue Morse proces start met 0.

Het voorbeeld van de Thue Morse reeks intrigeerde me. De traditionele manier om hiernaar te kijken is recursief, door de reeks te zien als opgebouwd als het resultaat van successieve expansie met behulp van de map

$$\begin{aligned} 0 &\mapsto 01 \\ 1 &\mapsto 10. \end{aligned}$$

Dit geeft:

$$1 \Longrightarrow 10 \Longrightarrow 1001 \Longrightarrow 10010110 \Longrightarrow 1001011001101001 \Longrightarrow \cdots$$

Of voor M':

$$0 \Longrightarrow 01 \Longrightarrow 0110 \Longrightarrow 01101001 \Longrightarrow 0110100110010110 \Longrightarrow \cdots$$

De specificatie laat zich rechtstreeks omzetten in een functioneel programma. Ter illustratie zal ik in dit stuk de taal Haskell gebruiken; deze brief is tevens een proeve van 'literate programming', het is in feite een becommentarieerd programma in Haskell.[1] De programma code is steeds de omkaderde tekst. Hier is het Haskell programma voor benaderingen van de Thue Morse reeks dat gebruik maakt van recursieve expansie:

```
expand ""       = ""
expand ('0':xs) = '0':'1': expand xs
expand ('1':xs) = '1':'0': expand xs

thuemorse 0 = "1"
thuemorse (n+1) = expand (thuemorse n)
```

[1] Zie `www.haskell.org` voor informatie over de taal.

De aanroep `thuemorse n` genereert the eerste 2^n tekens van de Thue Morse reeks M. We krijgen bij voorbeeld:

```
Main> thuemorse 0
"1"
Main> thuemorse 1
"10"
Main> thuemorse 2
"1001"
Main> thuemorse 3
"10010110"
Main> thuemorse 4
"1001011001101001"
Main> thuemorse 5
"10010110011010010110100110010110"
Main> thuemorse 6
"1001011001101001011010011001011001101001011010010110011010010110"
Main>
```

Corecursieve programma's voor M en M'

Dit is aardig, maar het volgt *niet* de uitleg van Jan Willem bij zijn lezing. De manier waarop Jan Willem de Thue Morse reeks in zijn praatje introduceerde was anders. De beginstring "1" is gegeven. Daarna kijk je steeds naar wat er al staat, swapt de nullen en enen, en schrijft het resultaat van de swap achter wat er al staat.

Omwisselen van de symbolen in een string van nullen en enen gaat als volgt:

```
swap ""       = ""
swap ('1': xs) = '0': swap xs
swap ('0': xs) = '1': swap xs
```

De procedure die Jan Willem gaf in zijn lezing is geen recursieve procedure maar een corecursieve. En inderdaad, het leidt tot een prachtig corecursief programma:

```
morse xs = swap xs ++ morse (xs ++ swap xs)

tm1 = '1' : morse "1"
```

Dit kan natuurlijk ook door eerst te swappen, en dan de geswapte string als parameter aan de hulpfunctie mee te geven. Dit geeft:

```
thue xs = xs ++ thue (xs ++ swap xs)

tm1a = '1' : thue "0"
```

Dit werkt, en het leuke is dat het *precies* de intuitieve uitleg volgt die Jan Willem gaf. Ik neem dit voorbeeld op als opgave in het boek van Kees Doets en mij over Redeneren en Programmeren met Haskell.[2] Ik ben op het ogenblik bezig met het afmaken en redigeren van de tekst. Het boek besteedt ruim aandacht aan het aanleren van wiskundige redeneertechnieken. In dat kader worden niet alleen inductieve bewijsmethoden geoefend, maar is er ook aandacht voor corecursie en coinductie. Ons boek is bij mijn weten het eerste tekstboek (bedoeld als inleiding redeneren en programmeren voor informatici) waarin recursie en corecursie, en inductie en coinductie, expliciet naast elkaar worden behandeld. Vandaar dat ik nu overal corecursie zie.

Overigens kent deze procedure ook een recursieve versie. De eerste 2^n tekens van M worden ook gegeven door `thuemorse2 n`:

```
thuemorse2 0 = "1"
thuemorse2 (n+1) = thuemorse2 n ++ swap (thuemorse2 n)
```

Jan Willem geeft in zijn *Wonderful Stream* voor jou[3] vier manieren om the Thue Morse reeks te genereren. De uitleg van zijn lezing is wat hij in zijn tekst manier (i) noemt. De definitie met behulp van $1 \to 10, 0 \to 01$ is manier (ii) uit zijn tekst. Als je met behulp van manier (ii) de hele reeks wilt genereren kan dat als volgt:

```
thmo n = thuemorse n ++ thmo (n+1)

tm2 = thmo 0
```

Dit is weer een voorbeeld van corecursief programmeren: de definitie van `thmo` gebruikt een parameter n, maar er is geen basisgeval, zoals bij een recursieve definitie.

Manier (iii) van Jan Willem definieert een functie $b(n)$ als het aantal enen in de binaire expansie van n, modulo 2. $b(n)$ geeft positie n in de reeks M'.

In de implementatie maken we gebruik van een hulpfunctie `binary` die een integer omzet in zijn binaire representatie, beschouwd als een lijst van nullen en enen. De functie $b(n)$ kijkt naar die lijst, filtert de enen eruit, en neemt de lengte van die nieuwe lijst, modulo 2:

[2] Kees Doets en Jan van Eijck, *Reasoning, Computation and Representation using Haskell*, onder submissie bij Cambridge University Press.

[3] Jan Willem Klop, *A Wonderful Stream for Jaco*, deze bundel.

```
binary x = reverse (bits x)
  where bits 0 = [0]
        bits 1 = [1]
        bits n = (rem n 2) : bits (quot n 2)

b x = length (ones x) `mod` 2
  where ones n = filter (==1) (binary n)
```

Dit geeft:

```
Main> map b [0..25]
[0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1]
```

Of als je liever een string hebt dan een reeks integers:

```
Main> map (intToDigit . b) [0..25]
"01101001100101101001011001"
```

Hier staat `intToDigit . b` voor de compositie van de functies `b` en `intToDigit`. De functie `intToDigit` zet een cijfer om in het corresponderende ASCII teken.

Haskell is een taal voor lazy list processing, dus je kunt ook de hele stroom genereren. Dit leidt tot de volgende definitie van de Thue Morse reeks (`swap` is nodig om M' om te zetten in M):

```
tm3 = swap (map (intToDigit . b) [0..])
```

Als ik het programma `tm3` op mijn computer opstart blijft het apparaat bits genereren tot ik het proces afbreek.

Manier (iv) van Jan Willem gebruikt de volgende functie ϵ:

$$\begin{aligned} \epsilon_0 &= 1 \\ \epsilon_{2n} &= \epsilon_n \\ \epsilon_{2n+1} &= 1 - \epsilon_n \end{aligned}$$

Hier is de Haskell versie:

```
epsilon 0 = 1
epsilon n | even n = epsilon (div n 2)
          | odd  n = 1 - epsilon (div (n-1) 2)
```

Wat is nu de bijbehorende functie voor M'? Simpelweg de functie ϵ' die je krijgt uit ϵ door de beginwaarde te swappen:

$$\begin{aligned} \epsilon'_0 &= 0 \\ \epsilon'_{2n} &= \epsilon'_n \\ \epsilon'_{2n+1} &= 1 - \epsilon'_n \end{aligned}$$

De implementatie wordt:

```
epsilon' 0 = 0
epsilon' n | even n = epsilon' (div n 2)
           | odd  n = 1 - epsilon' (div (n-1) 2)
```

Dit geeft:

```
Main> map epsilon [0 .. 100]
[1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,0,1,
1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,
1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,1,0,0,1,0]
Main> map epsilon' [0 .. 100]
[0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,1,0,
0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,
0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,0,1,1,0,1]
```

Een en ander leidt tot de volgende programma's voor M en M':

```
tm4 = map (intToDigit . epsilon) [0..]

tm4' = map (intToDigit . epsilon') [0..]
```

Overigens kan dit nog worden omgezet in een expliciete corecursie (vergelijk de opmerking van Jan Willem dat coinductie technieken hetzelfde kunnen klaarspelen als infinitair herschrijven):

```
h n = (intToDigit . epsilon) n : (h (n+1))

tm4a = h 0
```

Ik voeg er hier nog een manier aan toe die het idee van corecursie in (i) combineert met het expansie idee uit (ii). De functie `expand` voor het expanderen van strings nullen en enen is boven gegeven. De Thue Morse reeks M is het resultaat van 1 opschrijven, gevolgd door genereren van een string uit het zaadje "0". Genereren uit een zaadje `xs` geschiedt door dat zaadje op te schrijven, en het te laten volgen door de string die het resultaat is van genereren met zaadje `expand xs`. We krijgen dan het volgende programma:

```
grow xs = xs ++ grow (expand xs)

tm5 = '1': grow "0"
```

Als je de functie `grow` vergelijkt met de functie `thue`, dan zie je dat hier gebruik is gemaakt van het feit dat voor elke eindige binaire string `xs` geldt dat `expand xs` gelijk is aan `xs ++ (swap xs)`.

Maar het kan nog anders. Jan Willem geeft in paragraaf 2 van zijn paper wat hij een *self similar property* van M noemt. Dit valt direct in Haskell te programmeren, en het leidt tot ons zesde programma voor M.

```
tm6 = '1' : '0':  alternate (tail tm6) (tail (swap tm6))

alternate (x:xs) (y:ys) = x: y: alternate xs ys
```

De code voor het in elkaar vlechten van twee oneindige lijsten kan nog iets compacter:

```
alt (x:xs) ys = x: alt ys xs
```

En het hele programma kan nog wat verder worden gestileerd:

```
tm7  = '1' : tm7a
tm7a = '0' : alt tm7a (swap tm7a)
```

Dit is in feite het systeem van bewaakte recursievergelijkingen (*guarded recursive equations*) dat Jan Willem geeft in paragraaf 3 van zijn paper.

De corecursieve programmas `tm1`, `tm2`, `tm5`, `tm6` en `tm7` hebben last van wat bij functioneel programmeren een *space leak* wordt genoemd. Ze maken gebruik van string concatenatie en/of

string parameters, en de lengte van de strings die steeds als parameter worden meegegeven groeit exponentieel. Vandaar dat deze programma's er na enige tijd wegens geheugengebrek de brui aan geven. Na enige honderden regels nullen en enen zie je:

```
Main> tm1
"011010011001011010010110011O ...
...
00110100110010110100101100110
ERROR - Garbage collection fails to reclaim sufficient space
```

Hetzelfde gebeurt met de andere genoemde programma's. Dit euvel is inherent aan de definities, die immers een parameter gebruiken waarvan het geheugenbeslag exponentieel groeit. De definities `tm3` en `tm4` kennen dit probleem niet.

Proces specificaties van M en M'

Maar terug naar de definities en implementaties van M. Wanneer we bedenken dat M' het resultaat is van swappen van de nullen en enen in M, dan zien we dat we M en M' zeer elegant kunnen definiëren met behulp van simultane corecursie, als volgt:

```
m  = '1' : n
m' = '0' : k
n  = '0' : alt n k
k  = '1' : alt k n
```

Dit leidt direct tot een mooie proces-specificatie voor M en M':

$$\begin{aligned} M &= 1 \cdot N \\ M' &= 0 \cdot K \\ N &= 0 \cdot (N \,[]\, K) \\ K &= 1 \cdot (K \,[]\, N) \\ (b \cdot X) \,[]\, Y &= b \cdot (Y \,[]\, X) \end{aligned}$$

Hierbij staat b voor een willekeurige bit en X, Y voor een willekeurig proces.

Terzijde zij hier opgemerkt dat Jan Willem's exercitie om M met behulp van hernoemen en communicatie te definiëren me niet echt overtuigd heeft. Intuitief is de [] operatie op processen immers gemakkelijk met behulp van procesvergelijkingen te specificeren. De interactie tussen [] en $\cdot, +$ wordt dan gegeven door de volgende vergelijkingen (waarvan de eerste twee overbodig zijn als we [] alleen voor oneindige processen definiëren):

$$
\begin{aligned}
v \,[]\, X &= v \cdot X \\
(v \cdot X) \,[]\, w &= v \cdot w \cdot X \\
(v \cdot X) \,[]\, Y &= v \cdot (Y []X) \\
(X + Y) \,[]\, Z &= X[]Z + Y[]Z
\end{aligned}
$$

Ik ben benieuwd wat Jan Willem hiervan vindt. In elk geval denk ik dat het antwoord op zijn vraag 'Kan M worden gespecificeerd in procesalgebra met bewaakte recursievergelijkingen, maar zonder hernoemen?' bevestigend luidt.

De Toeplitz reeks T

In paragraaf 4 van zijn paper bespreekt Jan Willem de Toeplitz reeks T die je krijgt door de verschilreeks te nemen van M. De verschilreeks maken voor een binaire stroom gaat in Haskell als volgt:

```
difseq (x:y:zs) = dif y x : difseq (y:zs)
  where dif '0' '1' = '1'
        dif '1' '0' = '1'
        dif  _   _  = '0'
```

Merk hierbij op dat dit in feite verschilrijen oplevert modulo 2. Wanneer we niet modulo 2 rekenen krijgen we immers $0 - 1 = -1$ versus $1 - 0 = 1$.

De verschilrij van M' nemen modulo 3 levert een kwadraat-vrije (*square-free*) reeks op: er komen geen substrings van de vorm WW in voor. De verschil-operatie wordt nu:

```
difseq' (x:y:zs) = dif y x : difseq' (y:zs)
  where dif '0' '1' = '2'
        dif '1' '0' = '1'
        dif  _   _  = '0'
```

Dit geeft:

$$
\begin{aligned}
M' &= 01101001100101101001011001101001100101100110100101\ldots \\
\Delta M' \pmod 3 &= 10212010201210212012102010212010201210201021201210\ldots
\end{aligned}
$$

De reeks $\Delta M' \pmod 3$ is een transliteratie van de reeks

$$\phi M' = 02101202120102101201021202101202120102120210120102\ldots$$

die je krijgt door het morfisme $01 \mapsto 0,10 \mapsto 1,00 \mapsto 2,11 \mapsto 2$ los te laten op M. De implementatie:

```
phi ('0':'1':xs) = '0' : phi ('1':xs)
phi ('1':'0':xs) = '1' : phi ('0':xs)
phi ('0':'0':xs) = '2' : phi ('0':xs)
phi ('1':'1':xs) = '2' : phi ('1':xs)
```

Dit geeft:

```
Main> take 50 (phi m')
"02101202120102101201021202101202120102120210120102"
```

Die reeks is kwadraat-vrij, dus elke transliteratie ervan is dat ook. De transliteratie t die we nodig hebben om uit de verschilrij $\Delta M'$ (mod 3) de rij $\phi M'$ te krijgen waarvan Morse en Hedlund bewijzen dat ze kwadraat-vrij is, is $t = \lambda x(x + 2 \pmod 3)$. Overigens volgt hieruit meteen dat de reeksen ϕM en ΔM (mod 3) ook kwadraat-vrij zijn. Het is gemakkelijk te zien dat ϕM kan worden verkregen uit $\phi M'$ door verwisselen van nullen en enen.

De verschilreeks-definitie van T leidt tot het volgende programma voor T:

```
t1 = difseq m
```

Hiermee krijgen we:

```
Main> take 60 t1
"101110101011101110111010101110101011101010111011101110101011"
```

Uiteraard zijn de verschilreeksen modulo 2 die je krijgt uit M en M' identiek. Laat T' het resultaat zijn van omwisselen van de nullen en enen in T.

De verschilreeksen modulo 3 uit M en M' zijn niet identiek, maar het zijn transliteraties van elkaar: ze zijn in elkaar over te voeren door een swap van tweeën en enen.

De reeks T kan worden beschouwd als het resultaat van het toepassen van een alternatieve expansiefunctie:

```
expand2 ""       = ""
expand2 ('0':xs) = '1':'1': expand2 xs
expand2 ('1':xs) = '1':'0': expand2 xs
```

We hebben nu drie verschillende morfismen geïmplementeerd. Het wordt de moeite waard om een generatie-functie te definiëren met parameters voor het gebruikte morfisme en voor de beginwaarde:

```
generate :: (String -> String) -> String -> Int -> String
generate m init 0 = init
generate m init (n+1) = m (generate m init n)
```

Nu kun je

- M benaderen met behulp van `generate expand "1"`,
- M' benaderen met behulp van `generate expand "0"`,
- T benaderen met behulp van `generate expand2 "1"`.

Jan Willem merkt op dat de recursieve definities `toeplitz0`, `toeplitz1` en `toeplitz2` in de limiet dezelfde stroom opleveren, i.e., je kunt T ook benaderen met `generate expand2 "0"`.

```
toeplitz0 = generate expand2 "0"
toeplitz1 = generate expand2 "1"

toeplitz2 0 = "1"
toeplitz2 (n+1) | even n    = toeplitz2 n ++  "0" ++  toeplitz2 n
                | otherwise = toeplitz2 n ++  "1" ++  toeplitz2 n
```

Een proces specificatie van T

De stroom M heeft de eigenschap dat teken $M(2n)$ steeds verschilt van teken $M(2n+1)$ (wanneer we het eerste teken van de reeks aanduiden met $M(0)$). Dit volgt direct uit definitie (iv) van M. Dat betekent dat de verschilreeks modulo 2 voor M op alle *even* plaatsen een 1 heeft (wanneer we tellen vanaf 0).

M heeft ook de eigenschap dat $M(2n+2) - M(2n+1) = 1 - (M(n+1) - M(n))$. Dit kun je bij voorbeeld halen uit de definitie van M met behulp van ϵ. Maar dat wil zeggen dat $T(2n+1) = 1-T(n)$. En dat wil weer zeggen dat we, door de oneven posities uit de verschilreeks T te nemen, de reeks T' krijgen.

Wat hieruit volgt is dat T ontstaat door om en om een 1 en een element van T' te nemen. Dit geeft het volgende corecursieve programma voor T:

```
t2 = alt ones (swap t2) where ones = '1' : ones
```

Weer kunnen we dit zonder de swap doen, door T and T' met simultane corecursie te definiëren:

```
t  = alt ones t' where ones = '1' : ones
t' = alt zeros t where zeros = '0' : zeros
```

Het belang van deze simultane corecursie is dat het ons een fraaie proces-specificatie oplevert voor T en T':

$$\begin{aligned} O &= 1 \cdot O \\ Z &= 0 \cdot Z \\ T &= 1 \cdot (T' [] O) \\ T' &= 0 \cdot (T [] Z) \\ (b \cdot X) \; [] \; Y &= b \cdot (Y [] X) \end{aligned}$$

Hierbij staat b weer voor een willekeurige bit en X, Y voor een willekeurig proces.

Recursievergelijkingen voor de elementen van T

We kunnen nu een volgende vraag van Jan Willem beantwoorden. Wat zijn de recursievergelijkingen voor de functie τ die de elementen van T levert?

We maken gebruik van de functie ϵ voor M (hierboven geïmplementeerd als `epsilon`), en van rekenen modulo 2. Uit de definitie van T plus de definitie van ϵ krijgen we met rekenen modulo 2:

$$\begin{aligned} \tau(2n) &= \epsilon(2n+1) - \epsilon(2n) \pmod 2 \\ &= (1 - \epsilon(n)) - \epsilon(n) \pmod 2 \\ &= 1 \\ \tau(2n+1) &= \epsilon(2n+2) - \epsilon(2n+1) \pmod 2 \\ &= \epsilon(n+1) - (1 - \epsilon(n)) \pmod 2 \\ &= 1 - (\epsilon(n+1) - \epsilon(n)) \pmod 2 \\ &= 1 - \tau(n) \end{aligned}$$

Dit geeft ook direct antwoord op een andere vraag van Jan Willem. Alle staarten $t^n(T)$ zijn inderdaad verschillend. De reeks T is *niet* uiteindelijk periodiek. Hier is de implementatie van τ:

```
tau n | even n = 1
      | odd n  = 1 - tau (div (n-1) 2)
```

Dit leidt tot de volgende alternatieve implementatie van T:

```
t3 = map (intToDigit . tau) [0..]
```

Uiteraard vormen de recursievergelijkingen voor de functie τ' die T' oplevert het spiegelbeeld van die voor de functie voor T:

$$\begin{aligned} \tau'(2n) &= 0 \\ \tau'(2n+1) &= 1 - \tau'(n) \end{aligned}$$

Dit geeft:

```
tau' n | even n = 0
       | odd n  = 1 - tau' (div (n-1) 2)

t3' = map (intToDigit . tau') [0..]
```

Genererende functies voor M, M', T, T'

Een manier om oneindige reeksen te bestuderen die al sinds jaar en dag zeer vruchtbaar is gebleken is de volgende. Beschouw de elementen uit de reeks als de coëfficiënten $[a_0, a_1, a_2, ..]$ van een machtreeks voor een genererende functie $g(z)$:

$$g(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \cdots$$

Genererende functies werden al bestudeerd door Pascal, die aantoonde dat $(a+b)^n$ een genererende functie is voor het aantal combinaties van n dingen met k tegelijk genomen.

```
module Galois2

where

data Galois2 = Zero | One deriving (Eq, Ord, Enum, Read, Bounded)

instance Show Galois2
  where show Zero = "0"
        show One  = "1"

instance Num Galois2
  where
  Zero + One    = One
  One  + Zero   = One
  _    + _      = Zero
  One  * One    = One
  _    * _      = Zero
  negate        = id
  abs           = id
  signum        = id
  fromInteger n | even n = Zero
                | odd n  = One
```

Figuur 1: Een module voor GF(2).

In ons geval dienen we te rekenen modulo 2, hetgeen wil zeggen dat we de genererende functies beschouwen over het eindige lichaam (of Galois lichaam) GF(2). Dit is het lichaam van restklassen modulo 2, dus het heeft alleen de elementen 0 en 1, met optellen en vermenigvuldigen modulo 2.

Optellen modulo 2 komt neer op toepassen van de logische XOR functie, terwijl vermenigvuldigen modulo 2 hetzelfde is als de logische AND. Vermenigvuldigen van een bit met -1, modulo 2, verandert niets, want $1 \equiv -1 \pmod 2$, dus rekenkundige negatie is *niet* hetzelfde als logische negatie. Mijn bedoeling is om genererende functies voor M, M', T, en T' af te leiden die kunnen worden geïllustreerd met een implementatie. Daarbij heb ik een module nodig voor het Galois lichaam GF(2): zie Figuur 1. De bewerkingen op machtreeksen die we nodig hebben zijn optellen en vermenigvuldigen modulo 2. De module in Figuur 2 implementeert optellen en vermenigvuldigen op polynomen en machtreeksen.[4] Optellen van machtreeksen komt neer op coëfficiëntsgewijs optellen, vermenigvuldigen van machtreeksen is convolutie van de coëfficiënten: het n-de lid van het product van $[a_0, a_1, a_2, \ldots]$ en $[b_0, b_1, b_2, \ldots]$ wordt gegeven door $\Sigma_{k=0}^{n} a_k b_{n-k}$. Door nu te kijken naar machtreeksen over het Galois lichaam kunnen we rekenen modulo 2 met

[4] Nadere toelichting in het hoofdstuk over corecursie in ons eerder genoemde boek.

```
module Polynomials where

z :: Num a => [a]
z = [0,1]

infixl 7 .*
(.*) :: Num a => a -> [a] -> [a]
c .* []     = []
c .* (f:fs) = c*f : c .* fs

instance Num a => Num [a] where
  fromInteger c   = [fromInteger c]
  negate []       = []
  negate (f:fs)   = (negate f) : (negate fs)
  fs     + []     = fs
  []     + gs     = gs
  (f:fs) + (g:gs) = f+g : fs+gs
  fs     * []     = []
  []     * gs     = []
  (f:fs) * (g:gs) = f*g : (f .* gs + fs * (g:gs))
```

Figuur 2: Een module voor Polynoomreeksen.

machtreeksen implementeren.

We zullen nu zien dat de operatie [] van 'om en om nemen' in GF(2) een zeer natuurlijke interpretatie heeft als een operatie op machtreeksen modulo 2. Merk allereerst op dat kwadrateren modulo 2 gaat volgens de regel $(A+B)^2 = A^2 + B^2$. Immers, de term met $2AB$ valt weg omdat $2 \equiv 0 \pmod 2$. Maar dat betekent dat kwadrateren van een polynoomreeks modulo 2 als volgt gaat:[5]

$$(a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \cdots)^2 = a_0 + a_1 z^2 + a_2 z^4 + a_3 z^6 + \cdots \pmod 2.$$

Met andere woorden: de coëfficiëntenreeks $[a_0, a_1, a_2, a_3, ..]$ wordt omgezet in de reeks

$$[a_0, 0, a_1, 0, a_2, 0, a_3, 0, ..].$$

We kunnen dit illustreren aan de implementatie (in het onderstaande werk ik met een systeem waarin modules Galois2 en Polynomials geladen zijn):

```
TM> [One,One,One,One,One,One]
[1,1,1,1,1,1]
```

[5] Zie ook `http://mathworld.wolfram.com/Thue-MorseSequence.html`.

```
TM> [One,One,One,One,One,One]^2
[1,0,1,0,1,0,1,0,1,0,1]
```

Verschuiven van een polynoomreeks over een positie naar rechts, met invoegen van een 0 aan het begin, geschiedt door vermenigvuldigen met z. Immers:

$$z \cdot (a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \cdots) = 0 + a_0 z + a_1 z^2 + a_2 z^3 + a_3 z^4 + \cdots$$

Een en ander houdt in dat we vervlechten van reeksen A en B over GF(2) (het proces $A[]B$) kunnen modelleren als:

$$A^2 + zB^2.$$

Hier is een voorbeeld:

```
TM> [One,One,Zero,One]^2 + z * [Zero,Zero,One,Zero]^2
[1,0,1,0,0,1,1,0]
```

Met behulp hiervan kunnen we de procesdefinitie voor M en M' nu omzetten in het volgende simultane stelsel van genererende functies:

$$\begin{aligned} M(z) &= 1 + z \cdot N(z) \pmod 2 \\ M'(z) &= z \cdot K(z) \pmod 2 \\ N(z) &= z \cdot (N^2(z) + z \cdot K^2(z)) \pmod 2 \\ K(z) &= 1 + z \cdot (K^2(z) + z \cdot N^2(z)) \pmod 2 \end{aligned}$$

Hetzelfde kunnen we doen voor T en T'. Hierbij moet bedacht worden dat $\frac{1}{1-z}$ de genererende functie is voor de rij $[1, 1, 1, \cdots]$. Om en om nemen van de reeks $\frac{1}{1-z} = [1, 1, 1, \ldots]$ en de reeks $T'(z)$ geschiedt door $(\frac{1}{1-z})^2$ en $z \cdot T'^2(z)$ bij elkaar op te tellen, alles modulo 2. Om en om nemen van de reeks $[0, 0, 0, \cdots]$ en de reeks T geschiedt door $[0, 0, 0, \cdots]$ en $z \cdot T^2$ bij elkaar op te tellen, maar het resultaat hiervan is weer $z \cdot T^2$. We krijgen dus:

$$\begin{aligned} T(z) &= \left(\frac{1}{1-z}\right)^2 + z \cdot T'^2(z) \pmod 2 \\ T'(z) &= z \cdot T^2(z) \pmod 2. \end{aligned}$$

Invullen van de formule voor $T'(z)$ in die voor $T(z)$ en vereenvoudigen met behulp van $(1-z)^2 = 1 - z^2 \pmod 2$ geeft de volgende formule voor de genererende functie van T:

$$T(z) = \frac{1}{1-z^2} + z^3 \cdot T^4(z) \pmod 2.$$

Ter afsluiting geef ik de bijbehorende implementaties van de genererende functies voor M, M', T en T', in de vorm van een module die de modules voor het Galois lichaam en voor rekenen met machtreeksen importeert:

```
module TM where

import Galois2
import Polynomials

m  = One  : n
m' = Zero : k
n  = Zero : (n^2 + (Zero: k^2))
k  = One  : (k^2 + (Zero: n^2))

t  = ones^2 + (Zero : Zero : Zero : t^4) where ones = One : ones
t' = Zero : t^2
```

Dit geeft inderdaad weer de goede reeksen:

```
TM> take 32 m
[1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0]
TM> take 32 m'
[0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1]
TM> take 32 t
[1,0,1,1,1,0,1,0,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,0,1,0,1,1,1,0,1,0]
TM> take 32 t'
[0,1,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,1,0,1,0,0,0,1,0,1]
```

Omdat vermenigvuldiging van een genererende functie met $\frac{1-z}{z}$ een genererende functie voor de verschilrij oplevert, kan T ook worden gekarakteriseerd als $\frac{1-z}{z}M(z)$. Het effect van delen door z is dat de coëfficiëntenrij een positie naar links verschuift. T wordt dus gegenereerd als de staart van $(1-z)M(z)$:

```
TM> take 32 (tail ((1-z) * m))
[1,0,1,1,1,0,1,0,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,0,1,0,1,1,1,0,1,0]
```

Een paar open vragen over $\Delta M \pmod 3$

Laat H de verschilrij $\Delta M \pmod 3$ zijn. Kunnen we nu ook recursievergelijkingen geven voor de elementen van H? Een definitie van een functie h voor de elementen van H in termen van ϵ valt gemakkelijk te leveren:

$$\begin{aligned} h(2n) &= \epsilon(2n+1) - \epsilon(2n) \pmod 3 \\ &= (1 - \epsilon(n)) - \epsilon(n) \pmod 3 \\ &= 1 - 2\epsilon(n) \pmod 3 \\ &= 1 + \epsilon(n) \end{aligned}$$

$$\begin{aligned} h(2n+1) &= \epsilon(2n+2) - \epsilon(2n+1) \pmod 3 \\ &= \epsilon(n+1) - (1 - \epsilon(n) \pmod 3 \\ &= 2 + \epsilon(n+1) + \epsilon(n) \pmod 3 \end{aligned}$$

Dit leidt tot de volgende implementatie:

```
ha n | even n = (1 + epsilon (div n 2))
     | odd  n = (2 + (epsilon (m+1)) + (epsilon m)) `mod` 3
         where m = (div (n-1) 2)
```

Het is mij niet bekend of h ook rechtstreeks te definiëren valt, dus zonder gebruik te maken van een functie voor M.

Een andere (voor mij) open vraag is die naar formule voor een genererende functie voor H. Zoals boven reeds vermeld kan een genererende functie voor een verschilrij ΔA in het algemeen worden verkregen uit een genererende functie $g(z)$ voor A door vermenigvuldiging van $g(z)$ met $\frac{1-z}{z}$. Voor een genererende functie voor H dienen we te rekenen over het Galois lichaam GF(3), het lichaam van restklassen modulo 3. Dus, als we een genererende functie $M(z)$ zouden hebben voor M over dit lichaam, dan zou de genererende functie voor H er als volgt kunnen uitzien:

$$\begin{aligned} H(z) &= \frac{1-z}{z} M(z) \pmod 3 \\ &= \frac{1+2z}{z} M(z) \pmod 3. \end{aligned}$$

Het is me echter niet gelukt een genererende functie voor M over GF(3) te vinden. De moeilijkheid is dat de truc om $[a_0, a_1, a_2, \ldots]$ te transformeren in $[a_0, 0, a_1, 0, a_2, 0, \ldots]$ met behulp van kwadrateren alleen werkt in GF(2). In GF(3) hebben we: $(A+B)^2 = A^2 + 2AB + B^2$, en dit is in het algemeen niet gelijk aan $A^2 + B^2$.

Tot slot

Beste Jaco, ik ben aan het eind gekomen van mijn wetenschappelijke beschouwing. Rest een persoonlijk slot. Mijn jaren hier op het CWI kunnen worden onderscheiden in twee perioden: het pre-kantelingstijdvak, waarin ik ressorteerde in een cluster (toen nog 'afdeling') onder jouw bewind, en een periode daarna in het gekantelde tijdperk, waarin ik los van jou door het instituut gezweefd heb. Die tweede periode als vrij atoom heeft me de periode daarvoor retrospectief meer doen waarderen. Het is wijsheid achteraf, maar ik zie nu dat je indertijd buitengewoon goed voor je groepsleiders hebt gezorgd. Mijn besluit om je afdeling te verlaten was in het licht van de omstandigheden op dat moment wellicht onvermijdelijk, maar een wijzer en bezonnener iemand dan ik toen was had die stap misschien niet genomen. Wie zal het zeggen? Weet in elk geval dat ik met grote waardering afscheid van je neem. Het ga je goed.

Met vriendelijke groet,

Jan van Eijck