Werkexemplaar

SVP_typering van eindige automaten

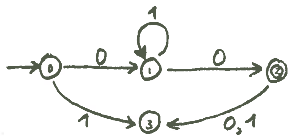
Maarten Fokkinga, 12 sept 1985

We beschouwen deterministische eindige acceptoren (automaten) en laten zien hoe de informele wiskundige typering van dergelijke dingen heel precies met de SVP-typering geformuleerd kan worden. Dit verhaal dient als een voorbeeld van de SVP-typering; niet als uitleg of definitie van die typering. Kennis van de SVP-typering is overigens niet nodig om dit verhaal te kunnen lezen en begrijpen.

* * *

1. Informele typering

We gaan ervan uit dat de lezer enigszins bekend is met het begrip deterministische eindige automaat die als acceptor wordt gebruikt. Hier is een voorbeeld:



Dit is een automaat met de vier toestanden ⓪, ①, ②, ③ waarvan ⓪ de zgn begintoestand is en ② de (enige) eindtoestand. De bedoeling is dat de automaat invoerrijen

van 0-en en 1-en al of niet accepteert. Daartoe wordt de automaat in de begintoestand gestart en steeds wordt één teken van de invoerrij afgesnoept waarbij de automaat over gaat naar een nieuwe toestand, aangegeven door de pijlen in het schema. Wanneer alle invoertekens zijn verwerkt en de toestand is een eindtoestand, dan is per definitie de invoerrij geaccepteerd en anders niet. De voorbeeldautomaat accepteert precies de tekenrijen $0 1^* 0$. Op het nut van dergelijke automaten gaan we nu niet in.

Een wiskundige beschrijving van dergelijke acceptoren luidt veelal als volgt.

(1) " Een (deterministische eindige) <u>acceptor</u> A is een viertal $\langle Q, q_0, t, E \rangle$
waarbij: Q een toestandsverzameling is,
q_0 een element van Q is (de begintoestand),
t een transitie funktie: $(Q, int \to Q)$, en
E een deelverzameling van Q is (de zgn. eindtoestanden).
"

De informele interpretatie van t is ~~dat wanneer~~ als volgt. Zij de automaat in toestand q ~~is~~ en zij i ~~is~~ het eerste teken op de invoerrij dat verwerkt moet worden. ~~dat~~ Dan is $t(q,i)$

Alfabet precies invoeren of afwezigheid daarvan motivere. Zo roept 't problemen op.

de nieuwe toestand waarin de automaat overgaat. We hebben voor het gemak aangenomen dat de invoertekens getallen zijn; dit is te zien aan het type van de transitiefunktie t.

Bij onze voorbeeldautomaat A kunnen we nemen:

(2a) $Q = \{0, 1, 2, 3\}$ (of zelfs Q = gehele getallen (<u>int</u>))

$q_0 = 0$

$t = \lambda q, x.\ q=0 \wedge x=0 \rightarrow 1;$

$q=0 \wedge x=1 \rightarrow 3;$

$q=1 \wedge x=1 \rightarrow 1;$

...

$E = \{3\}$ (of $E = [3]$ als verzamelingen als lijsten worden gerepresenteerd).

Maar er zijn ook andere keuzen mogelijk. Voor de accepterende werking van de automaat doet de keuze van de verzameling Q er niet zoveel toe, ja zelfs die doet er niets toe. Dat blijkt ook al uit het diagram: toestanden zijn daar louter rondjes op papier, verschillende rondjes zijn verschillende toestanden. We hadden voor Q bijvoorbeeld ook <u>string</u>s kunnen nemen, en wel: "begin", "halfweg", "eind", "fout". Preciezer:

(2b) Q = <u>string</u>

q_0 = "begin"

$t = \lambda q, x.$ q= "begin" $\wedge$ x=0 $\rightarrow$ "halfweg"

q= "begin" $\wedge$ x=1 $\rightarrow$ "fout"

q= "halfweg" $\wedge$ x=0 $\rightarrow$ "eind"

...

E = ["eind"]

De accepterende werking van een automaat $A = \langle Q, q_0, t, E \rangle$ wordt als volgt gedefinieerd.

(3) " Definieer allereerst met inductie naar de lengte van de invoerrij X welke invoerrijen vanuit welke toestand geaccepteerd worden; $f(q, X) =$ <u>true</u> als X geaccepteerd wordt vanuit toestand q, en <u>false</u> anders.

$$f(q, []) = \text{'}q \in E\text{'}$$
$$f(q, x{:}X) = f(t(q,x), X)$$

Zet dan

$$Accept_A(X) = f(q_0, X)$$

"

Dus voor de voorbeeldautomaat A geldt: $Accept_A([0,1,1,1,0]) =$ <u>true</u> en $Accept_A([0,1,0,1]) =$ <u>false</u>. Dit is inderdaad aan de hand van de definities in (2a) en (2b) na te rekenen.

Tot zover de informele behandeling van acceptoren. We gaan nu over tot de formele, i.e. SVP-getypeerde, beschrijving.

2. De SVP getypeerde formulering

We geven nu nogmaals weer wat zojuist al onder (1), (2) en (3) is uitgedrukt. We vertellen dus niets nieuws, maar gebruiken alleen een getypeerde notatie.

(1') acceptor : <u>tp</u>
== < Q : <u>tp</u>
, q_0 : Q
, t : (Q, int → Q)
, E : Q <u>list</u>
>

Opmerkingen.

a. Vanwege het == -teken moeten we in het vervolg overal waar 'acceptor' staat de definiërende expressie lezen, i.e. <Q: tp, --- >. (Had er een =-teken gestaan dan was 'acceptor' een nieuwe type-identifier geweest waarvan niet bekend was, t.a.v. de type-controle, dat het staat voor een groep met vier komponenten.)

b. ~~Omdat~~ Merk op dat de identifiers Q, q_0, t, E twee rollen vervullen. Ten eerste dient Q in de tweede en volgende komponent als naam voor (de waarde van) de eerste komponent. Net zo voor q_0, t en E; maar toevallig komen q_0, t en E verder niet meer voor en hadden we ze wel weg kunnen laten voor wat deze rol betreft (net zo als we voor het type van t niet (q: Q, i: int → Q) maar (Q, int → Q) hebben geschreven). Ten tweede zullen we die namen straks gebruiken als selectiemiddel: net zo als de field-identifiers van Pascal-records.

c. Merk op dat uit de vergelijking van (1') met (1) blijkt dat (1) géén definitie van een acceptor A is, maar van het type 'acceptor'. De 'A' in de definitie (1) is dan ook volkomen misplaatst: we hadden beter ná definitie (1) kunnen vermelden dat we voortaan A gebruiken als naam voor acceptoren. (A is een meta-variabele over acceptoren; identifier A heeft, by default, type 'acceptor').

Nu de voorbeeldautomaat:

(2a) A : acceptor
    = < int
        , 0
        , λq: int, x: int. q=0 ∧ x=0 → 0;
                                q=0 ∧ x=1 → 1;
                                ...
        , [3]
      >

Dit is een goed-getypeerde definitie:

- de eerste komponent, int, is van type tp;
- de tweede komponent, 0, is van type Q waarbij Q staat voor de eerste komponent;
- de derde komponent, (λq: int, x: int. ....), is van type (Q, int → Q) waarbij Q en q_0 staan voor de eerste en tweede komponent (maar q_0 komt "toevallig" niet voor!)
- etc.

Dus de groep als geheel heeft type acceptor.

We hadden A ook anders kunnen definiëren:

(2b) A' : acceptor
    = < string
        , "begin"
        , λq: string. x: int. q="begin" ∧ x=0 → "halfweg";

        , ["eind"]
      >

Ook hier is de definierende expresie inderdaad van type acceptor: de eerste komponent, ~~...~~ string, is van type tp, de tweede komponent, "aap", is van type Q waarbij Q staat voor de eerste komponent, etc etc. Dus de definitie van acceptor in (1') laat nog verschillende representaties van acceptoren toe, zoals (2a) en (2b).

Nu tenslotte de accepterende werking van acceptoren.

```
(3')  Accept
        : (acceptor  →  ( int list  →  bool))
        = (λ A : acceptor.
                δ  Q == A·Q ;   q0 == A·q0 ;   t == A·t ;   E == A·E
                ;  f : ( Q, intlist  → bool)
                     = (λq: Q,  X: int list .
                              if  X = []  then   "q ∈ E"
                                                 else   f (t (q, (hd X)), tl X)
                       )
                •  λX: int list.  f (q0, X)
      )
```

Ook deze definitie is, uiteraard, goed getypeerd: we zien onder andere dat A van type acceptor is, dus A·Q van type tp, dus is (Q, intlist → bool) een type; voorts is q_0 van type Q en dus is de resultaatfunktie (λX: intlist. f (q_0, X)) van type (intlist → bool), zoals al aangekondigd in het type van Accept.

De verdienste van de SVP-typering zit niet in het feit dat informele definities zoals (1), (2) en (3) óók formeel genoteerd kunnen worden, zie (1'), (2a' 2b') en (3).

Nee, de verdienste van de SVP-typering zit in het feit dat sommige definities die informeel wél mogelijk zijn nu juist niet mogelijk zijn. Informeel zouden we bijvoorbeeld de vraag kunnen stellen: ~~of~~

(4) is 0 de begintoestand van de in (2) gedefinieerde automaat A ?

En het antwoord is: ja. Maar de formulering

(4') $0 = A \cdot q_0$ (A gedefinieerd in (2a))

is niet goed SVP-getypeerd. Immers A is van type acceptor, dus $A \cdot q_0$ is van type Q waarbij Q staat voor de eerste komponent van A, dus $A q_0$ is van type A·Q ; en volgens de definitie van acceptor is van A·Q niets anders bekend dan dat het een type is, A·Q : tp. De nul is van type int. Dus de typen van de linker- en rechterkant van (4') zijn

int respectievelijk A·Q (beide van type tp).

Aangezien dit verschillende expressies zijn, ~~zijn ze~~ terwijl de ==-operator aan beide zijden gelijke typen ~~...~~ eist, is formule (4') fout getypeerd.

Het is geenszins onplezierig dat vraag (4') niet goed getypeerd is. Want zou (4') wel O.K. zijn, dan zou dat natuurlijk ook het geval zijn met

(4'') $\quad 0 = A'.q_0 \quad$ (A' gedefinieerd in (2b'))

En nu is de waarde van $A'.q_0$ inderdaad géén integer maar een string. Door de SVP-typering zijn de feitelijke representaties, zoals in (2a) en (2b') vastgelegd, afgeschermd: van de representatie van een automaat kan alleen gebruik gemaakt worden voor zover het type acceptor dat specificeert; de definitie van Accept in (3') is daar een voorbeeld van, de ~~vergelijkingen~~ testen (4') en (4'') zijn daar tegenvoorbeelden van.

Overigens kunnen we wel de vraag stellen of

(5) $\quad A\cdot q_0 = A\cdot q_0 \quad$ (beide van type $A\cdot Q$)
(5') $\quad A'\cdot q_0 = A'\cdot q_0 \quad$ (beide van type $A'.Q$)
(5'') $\quad A\cdot q_0$ "$\in$" $A\cdot E \quad$ ($A\cdot E$ van type $A\cdot Q$-list)

maar niet

(5''') $\quad A'\cdot q_0 = A\cdot q_0$

want $A'\cdot q : A'\cdot Q$ terwijl $A\cdot q_0 : A\cdot Q$ en $A'\cdot Q$ verschilt van $A\cdot Q$ (louter omdat A en A' verschillende identifiers zijn) dus zijn de typen niet gelijk; en dat is wel vereist bij een =-test. Ook hier weer is dank zij de SVP-typering de representatie afgeschermd.

3 Tot besluit

Middels de SVP-typering kunnen representatie (of implementatie-)keuzen afgeschermd worden: ook al ken je die keuzen, je kan er toch geen gebruik van maken want dat zou type-foute programma's geven. Zo'n syntactisch afgedwongen afscherming heeft een groot voordeel: wanneer om wat voor reden dan ook de representatie wordt gewijzigd, dan is gegarandeerd dat niets in het gebruikersprogramma mee-gewijzigd hoeft worden, eenvoudigweg omdat in het gebruikersprogramma niet van die representatiekeuzen gebruik gemaakt kan worden. (Natuurlijk moet de abstracte betekenis niet gewijzigd worden; die moet in stand gehouden worden en alleen de representatie (dus niet de interpretatie!) mag veranderen. De SVP-typering biedt hiervoor geen of nauwelijks enige steun; er zijn typeringen waarbij ook hier type-korrektheid garandeert dat de betekenis niet ~~ver~~verandert, nml.de Intuitionistische Type Theorie van Martin-Löf.)

Voorts nog de volgende, belangrijke, opmerking. Wanneer je in de SVP-getypeerde teksten de accepterende werking van een ~~automaat~~ acceptor tot in detail probeert uit te schrijven, dan loop je vast bij "$q \in E$" <u>tenzij</u> je veronderstelt dat de gelijkheidsoperatie $=$ op ieder type is gedefinieerd. Wanneer dit niet het geval is moet je de definitie van ~~automaten~~ acceptoren aanpassen: in plaats van een lijst E van eindtoestanden kun je beter een functie E geven die test of een toestand een eindtoestand is. De benodigde gelijkheid hoeft dan niet "geexporteerd" te worden, maar is alleen intern in de definierende expressies van ~~automaten~~ acceptoren nodig. Merk op dat het niet zinnig is ~~een~~ om te veronderstellen dat er voor ieder type een gelijkheidstest bestaat! Immers zo'n standaard-operatie kan dan alleén maar als betekenis hebben dat --bij zelf gedefinieerde typen zoals acceptor -- gelijkheid van de representatie wordt getest. Dat is in het algemeen een te sterke eis (er kunnen best verschillende representaties van eenzelfde toestand zijn!) en bovendien niet altijd mogelijk (we hadden ook best <u>functies</u> als representaties van toestanden kunnen geven en gelijkheid van functies is niet

beslisbaar!). We concluderen hieruit dat de informele typering van acceptoren, zoals verwoord in definitie (1), slecht gekozen is. (En dat is in dit verhaal dank zij de SVP-typering aan het licht gebracht!)