COW

```
                                        /;    ;\
                                    __  \\____//
                                   /{_\_/   `'\____
                                   \___   (o)  (o  }
       ______________________________/          :--'
   ,-,'`@@@@@@@@@     @@@@@@         \_    `__\
  ;:(  @@@@@@@@@       @@@             \___(o'o)
  :: )  @@@@          @@@@@@        ,'@@(  `===='    Moo!
  :: : @@@@@:          @@@@         `@@@:
  :: \  @@@@@:       @@@@@@@)    (  '@@@'
  ;; /\      /`,    @@@@@@@@@\   :@@@@@)
  ::/  )    {_----------------:  :~`,~~;
 ;;'`; :   )                  :  / `; ;
;;;; : :   ;                  :  ;  ; :
`'`' / :  :                   :  :  : :
    )_ \__;      ";"          :_ ;  \_\       `,','
    :__\  \    * `,'*         \  \  :  \   *   8`;'*  *
 ` ` `  `^' ` ` \ :/ ` ` ` ` `  `^'  `-^-' ` \v/ `:  \/
```

Werkstuk ‘Semantiek in het wild’

Semantiek en Correctheid (T3)

Alex van Oostenrijk (0153729)
Martijn van Beek (0343080)

Inhoud

Samenvatting

In dit verslag wordt de semantiek van de taal COW beschreven. COW is een grappige programmeertaal ontwikkeld binnen de *Esoteric Programming Languages Ring*, waarbij alle sleutelwoorden in de taal een variant van *MOO* zijn. Wij beschrijven de natuurlijke en structurele operationele semantiek van de taal. Ook laten wij zien dat de taal COW Turing compleet is.

Taalspecificatie

In de onderstaande tabel zijn alle beschikbare instructies in de taal Cow weergegeven volgens [Heber]. Het is belangrijk om op te merken dat de syntax van Cow geen enkele volgorde van de instructies afdwingt. Ze kunnen in willekeurige volgorde achter elkaar worden geplaatst om een geldig Cow-programma te vormen. Dit heeft vooral nare gevolgen voor lussen, omdat deze bestaan uit twee instructies (`MOO` en `moo`) die bij elkaar horen. In de rest van dit verslag bespreken wij de problemen die optreden in de semantiek en hoe wij deze oplossen.

Code	Instructie	Betekenis
0	`moo`	Dit commando gaat samen met `MOO`. Het effect is dat de computer terug in de code zoekt naar een bijbehorend `MOO` commando, en daar de uitvoering hervat. Bij het zoeken slaat het de instructie die er direct vóór staat over (zie `MOO`).
1	`mOo`	Verplaatst de huidige geheugenpositie 1 blok terug.
2	`moO`	Verplaatst de huidige geheugenpositie 1 blok vooruit.
3	`mOO`	Voert de inhoud van het huidige geheugenblok uit als een instructie. Het commando wordt gekozen op basis van de instructiecode (e.g. 2 = `moO`). Een onjuiste code (kleiner dan 0, of groter dan 11) stopt het programma. De code 3 is ook niet toegestaan, omdat dit een oneindige lus zou veroorzaken.
4	`Moo`	Indien het huidige geheugenblok 0 bevat, dan leest deze instructie één teken van de invoer en plaatst dit als getal (ASCII-waarde) in het huidige geheugenblok. Indien de waarde van het huidige geheugenblok niet 0 is, dan zet de instructie deze waarde als ASCII-karakter op de uitvoer.
5	`MOo`	Verlaagt de waarde van het huidige geheugenblok met 1.
6	`MoO`	Verhoogt de waarde van het huidige geheugenblok met 1.
7	`MOO`	Indien de waarde van het huidige geheugenblok 0 is, dan slaat deze instructie het volgende commando over en hervat de executie na het bijbehorende `moo` commando. Indien de waarde van het huidige geheugenblok niet 0 is, dan gaat de executie verder met het volgende commando.
8	`OOO`	Maakt de waarde van het huidige geheugenblok gelijk aan 0.
9	`MMM`	Als het register geen waarde bevat, dan krijgt het de waarde van het huidige geheugenblok. Anders wordt de waarde van het register in het huidige geheugenblok geplaatst en wordt het register leeggemaakt.
10	`OOM`	Plaatst de waarde van het huidige geheugenblok op de uitvoer.
11	`oom`	Leest een getal van de invoer en plaatst het in het huidige geheugenblok.

Syntax

De grammatica van de taal COW is als volgt gespecificeerd.

S ::= moo | mOo | moO | mOO | Moo | MOo | MoO | MOO | OOO | MMM | OOM | oom | S_1 S_2.

Daarbij staat *S* voor *statement*. Er is géén teken nodig om statements in een compositie van elkaar te scheiden. In COW zijn alle statements drie tekens lang zodat een parser altijd weet waar een statement begint en eindigt.

Aangezien `MOO` en `moo` samen een while-lus vormen, ligt het voor de hand ze als één syntactische constructie op te nemen. Dat is echter niet mogelijk, omdat de instructie `mOO` losse `MOO`'s en `moo`'s kan produceren, zodat (halve) while-lussen dynamisch kunnen worden gemaakt. Een programma kan ook best eindigen terwijl het nog in een while-lus zit!

State

De state is een 6-tupel s = (M, p, r, V, W, l) waarbij geldt:

- M is een oneindige lijst $m_0, m_1, \ldots$ met $m_i \in \mathbb{Z}$ en stelt de geheugeninhoud voor. Aan het begin van de uitvoering van een programma is deze lijst gevuld met nullen.
- $p \in \mathbb{N}$ is de geheugenteller. Deze wijst naar het actieve geheugenelement $m_p \in M$. De geheugenteller is initieel 0 en kan nooit kleiner dan 0 worden.
- $r \in \mathbb{Z} \cup \varepsilon$ is het register. Het register is initieel leeg (ε).
- V is een eindige lijst invoerwaarden $v_0,\ldots,v_n$ met $v_i \in \mathbb{Z}$ en stelt de invoerwaarden van het toetsenbord naar het programma voor. Aan het begin van de uitvoering van een programma geldt $n \geq 0$.
- W is een eindige lijst uitvoerwaarden $w_0, \ldots, w_n$ met $w_i \in \mathbb{Z}$ en stelt de uitvoerwaarden van het programma voor. De lijst W is initieel leeg (n = 0).
- l is het *nesting level*, dat nodig is voor zoeken in while-loops (zie natuurlijke semantiek). In het begin van het programma is het nesting level 0.

We kiezen om invoer en uitvoer in de state te modelleren, zodat de semantiek van een programma met een gegeven invoer (als een rij getallen) bepaald kan worden. Bovendien speelt de invoer van `stdin` en de uitvoer naar `stdout` een wezenlijke rol in COW. We willen in staat zijn om de executie van een programma aan de hand van de eindstate te beoordelen; deze eindstate bevat dan de invoer- en uitvoerrijen na volledige uitvoering van het programma.

Benadering van de State

Omdat de state complex is en niet een functie Var $\rightarrow$ Z zoals bij de taal While [Nielson92], moeten wij een nieuwe methode bedenken om de state te benaderen en te wijzigen. Deze methode kan bestaan uit een aantal functies met signatuur State $\rightarrow$ State (in feite soortgelijk aan de substitutie-operatie [ ] die in [Heber] wordt gebruikt).

Voor het uitvoeren van de instructies van COW is toegang nodig tot alle elementen in de state, en niet slechts tot de geheugeninhoud. Voor iedere wijziging is dus een functie nodig. Een voorbeeld van een functie die de instructieteller met 1 verlaagt:

$$\delta(m,p,r,v,w,l) = \begin{cases} (m,p\text{-}1,r,v,w,l) & \text{als } p > 0 \\ (m,p,r,v,w,l) & \text{anders} \end{cases}$$

Natuurlijk zijn er prettiger notaties te verzinnen:

$$\delta(s) = \begin{cases} s[p \rightarrow p\ \text{-}1] & \text{als } p > 0 \\ s & \text{anders} \end{cases}$$

Dit kan omdat de functies meestal maar op één onderdeel van de state tegelijk werken. De definities van de functies die de huidige geheugenwaarde verhogen, verlagen of op nul zetten, getallen toevoegen aan de uitvoerrij of lezen uit de invoerrij zijn flauw.

We moeten oppassen dat geen werk in de functies gedaan wordt dat eigenlijk in de specificatie van de natuurlijke semantiek gedaan had moeten worden. Er zou een functie kunnen komen die de instructie `Moo` evalueert:

4	`Moo`	Indien het huidige geheugenblok 0 bevat, dan leest deze instructie één teken van de invoer en plaatst dit als getal (ASCII-waarde) in het huidige geheugenblok. Indien de waarde van het huidige geheugenblok niet 0 is, dan zet de instructie deze waarde als ASCII-karakter op de uitvoer.

Dit kan omdat alle benodigde informatie in de state opgeslagen is, maar het effect van `Moo` kan ook uitgedrukt worden in het tableau voor natuurlijke semantiek, wat eleganter en overzichtelijker is. We kiezen daarom om alleen erg eenvoudige functies te bouwen en zullen deze niet bij naam noemen, maar liever een syntax gebruiken als:

$$\langle \text{Moo}, S\rangle \to s[m_p \to v_0, v \to \text{tail}(v)] \quad \text{als } m_p = 0$$

Men kan er ook een s bij denken, zoals in $s.m_p$, waarmee de notatie dan lijkt op die van records in de programmeertaal *Clean*. Voor operaties op lijsten (rijen) werken wij met eenvoudige operatoren en functies uit Clean, te weten:

Functie	**Betekenis**	**Voorbeeld**
++	Lijstconcatenatie	$[1,2]$ ++$[3,4] \to [1,2,3,4]$
head	Eerste element van een lijst	head $[1,2,3] \to 1$
tail	Lijst minus eerste element	tail $[1,2,3] \to [2,3]$

Natuurlijke Semantiek

De natuurlijke semantiek van COW is gemakkelijk op te schrijven, ware het niet dat de instructie `mOO` (3) roet in het eten gooit. Omdat wij in de syntax niet in staat waren om while-lussen als één constructie te zien (maar in plaats daarvan als twee helften), komen wij nu in de problemen.

Triviale instructies

Alle instructies behalve 0, 3 en 7 zijn triviaal om te maken, evenals de compositie van statements:

$$[\text{comp}] \quad \frac{\langle S1, s\rangle \to s', \quad \langle S2, s'\rangle \to s''}{\langle S_1\ S_2, s\rangle \to s''}$$

1	$[\text{mOo}^{p>0}]$	$\langle \text{mOo}, s\rangle$	$\to s[p \to p - 1]$	als $p > 0$
1	$[\text{mOo}^{p=0}]$	$\langle \text{mOo}, s\rangle$	$\to s$	als $p = 0$
2	[moO]	$\langle \text{moO}, s\rangle$	$\to s[p \to p + 1]$	
4	$[\text{Moo}^{in}]$	$\langle \text{Moo}, s\rangle$	$\to s[m_p \to v_0, v \to \text{tail}(v)]$	als $m_p = 0$
4	$[\text{Moo}^{out}]$	$\langle \text{Moo}, s\rangle$	$\to s[v \to v + m_p]$	als $m_p \neq 0$
5	[MOo]	$\langle \text{MOo}, s\rangle$	$\to s[m_p \to m_p - 1]$	
6	[MoO]	$\langle \text{MoO}, s\rangle$	$\to s[m_p \to m_p + 1]$	
8	[OOO]	$\langle \text{OOO}, s\rangle$	$\to s[m_p \to 0]$	
9	[MMM]	$\langle \text{MMM}, s\rangle$	$\to s[r \to m_p]$	als $r = \varepsilon$
9	[MMM]	$\langle \text{MMM}, s\rangle$	$\to s[m_p \to r, r \to \varepsilon]$	als $r \neq \varepsilon$

10	[OOM]	$\langle OOM, s\rangle$	$\rightarrow s[w \rightarrow w \mathbin{+\!+} m_p]$
11	[oom]	$\langle oom, s\rangle$	$\rightarrow s[m_p \rightarrow v_0, v \rightarrow tail(v)]$

Lus-instructies

Voor 0 en 7 (`moo` en `MOO`) moeten wij meer moeite doen, vanwege het bestaan van de halve while-loops. We gaan dit oplossen met behulp van een *environment*, waarin wij de body van zo'n while-loop opslaan (inclusief de bijbehorende `MOO` en `moo`). Het environment *env* is een (initieel lege) lijst van statements (meestal composities). Wanneer een lus begint met MOO, dan plaatst MOO de statements in de lus in het environment. Het einde van een lus, moo, haalt de statements er weer uit. Omdat lussen genest kunnen zijn, is het nodig dat het environment een lijst van statements bevat. Een voorbeeld van een environment dat de inhoud van twee geneste lussen bevat is:

env = [[MOO MOO OOO moo moo], [MOO OOO moo]]

Merk op dat het beoogde effect ook bereikt had kunnen worden met *continuations* (zoals deze voor de invoering van het *break*-statement in de taal While [Nielson92] gebruikt werden). Met gebruik van continuations is echter nog steeds een nesting flag (zie hieronder) nodig, zodat wij niet denken dat er dan minder regels nodig zijn in de natuurlijke semantiek.

Lusexecutie

Indien $m_p \neq 0$, dan moeten alle instructies die op `MOO` volgen worden uitgevoerd totdat de bijpassende `moo` wordt gevonden. Omdat wij bij het aantreffen van een `moo` niet zomaar terug in de code kunnen kijken, lossen wij dit op met een environment. Bij het ingaan van een lus bewaren wij alle op `MOO` volgende statements in het environment, en bij de uitvoering van `moo` halen we alle statements er weer af. Dit proces kan genest plaatsvinden, met meerdere stukken code in het environment.

$$[MOO^{loop}] \qquad \frac{push(MOO\ S, env) \vdash \langle S, s\rangle \rightarrow s'}{env \vdash \langle MOO\ S, s\rangle \rightarrow s'} \qquad \text{als } m_p \neq 0$$

$$[moo^{loop}] \qquad \frac{pop(env) \vdash \langle S, s'\rangle \rightarrow s'}{env \vdash \langle moo, s\rangle \rightarrow s'} \qquad \text{met } top(env) = S$$

De toegang tot het environment wordt geregeld met de functies *push*, *pop* en *top*. Deze zijn als volgt gedefinieerd:

push:	$S \times env \rightarrow env$	$= env \mathbin{+\!+} S$
pop:	$env \rightarrow env$	$= tail(env)$
top:	$env \rightarrow S$	$= head(env)$

Het environment is een lijst van statementlijsten. De hulpfuncties *++*, *head* en *tail* zijn gedefinieerd zoals in de programmeertaal *Clean*.

Voorwaarts zoeken

Volgens de specificatie van `MOO` moet deze instructie alle volgende instructies overslaan (tot de bijbehorende `moo`) indien $m_p = 0$. Hierbij moet de instructie die direct op MOO volgt altijd worden overgeslagen.

Om de laatste voorwaarde te laten werken vervangen wij `MOO` direct door `MOO2` en slaan de eerstvolgende instructie over. We maken gebruik van de compositie van statements in de vijf natuurlijke semantiekregels voor `MOO`. Om te zorgen dat een het zoeken pas stopt bij de bijpassende `moo` (en niet bij een geneste `moo`) houden wij het *nesting level* in de state bij (als l).

$$[\text{MOO}^{\text{begin search}}] \quad \frac{\text{env} \vdash \langle \text{MOO2 } S_2, s\rangle \to s'}{\text{env} \vdash \langle (\text{MOO } S_1)\ S_2, s\rangle \to s'} \quad \text{als } m_p = 0$$

$$[\text{MOO}^{\text{search}}] \quad \frac{\text{env} \vdash \langle \text{MOO2 } S_2, s\rangle \to s'}{\text{env} \vdash \langle (\text{MOO2 } S_1)\ S_2, s\rangle \to s'} \quad S_1 \neq \text{MOO} \ \&\ S_1 \neq \text{moo}$$

$$[\text{MOO}^{\text{begin nest}}] \quad \frac{\text{env} \vdash \langle \text{MOO2 } S_2, s[l \to l + 1]\rangle \to s'}{\text{env} \vdash \langle (\text{MOO2 MOO})\ S_2, s\rangle \to s'}$$

$$[\text{MOO}^{\text{end nest}}] \quad \frac{\text{env} \vdash \langle \text{MOO2 } S_2, s[l \to l - 1]\rangle \to s'}{\text{env} \vdash \langle (\text{MOO2 moo})\ S_2, s\rangle \to s'} \quad \text{als } l > 0$$

$$[\text{MOO}^{\text{end search}}] \quad \text{env} \vdash \langle (\text{MOO2 moo})\ S_2, s\rangle \to s \quad \text{als } l = 0$$

Voorbeeld van geneste lus

We bepalen de natuurlijke semantiek van het volgende programma, uitgaande van een state waarin $m_p = 1$. Het programma bevat twee geneste loops, die beide precies één keer uitgevoerd moeten worden (want $m_p = 1$).

$$\left.\begin{array}{l}\texttt{MOO}\\ \texttt{OOM}\\ \left.\begin{array}{l}\texttt{MOO}\\ \texttt{OOO}\\ \texttt{moo}\end{array}\right\}\text{inner loop}\\ \texttt{moo}\end{array}\right\}\text{outer loop}$$

$$\dfrac{\dfrac{[\text{outer}] \vdash \langle \text{OOM}, s\rangle \to s[w \to w \mathbin{++} m_p] \qquad \dfrac{\dfrac{[\text{outer,inner}] \vdash \langle \text{OOO}, s\rangle \to s[m_p \to 0] \qquad \dfrac{\dfrac{\dfrac{[\text{outer}] \vdash \langle \text{MOO2 moo moo}, s\rangle \to s}{[\text{outer}] \vdash \langle \text{MOO OOO moo moo}, s\rangle \to s}}{[\text{outer,inner}]\ \langle \text{moo}, s\rangle \to s} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\ ] \vdash \langle \text{MOO2 moo}, s\rangle \to s}{[\ ] \vdash \langle \text{MOO2 moo}, s[l \to l - 1]\rangle \to s}}{[\ ] \vdash \langle \text{MOO2 moo moo}, s\rangle \to s}}{[\ ] \vdash \langle \text{MOO2 OOO moo moo}, s[l \to l+1]\rangle \to s}}{[\ ] \vdash \langle \text{MOO2 MOO OOO moo moo}, s\rangle \to s}}{[\ ] \vdash \langle \text{MOO OOM MOO OOO moo moo}, s\rangle \to s}}{[\text{outer}]\ \langle \text{moo}, s\rangle \to s}}{[\text{outer,inner}]\ \langle \text{moo moo}, s\rangle \to s}}{[\text{outer,inner:=[MOO OOO moo moo]}] \vdash \langle \text{OOO moo moo}, s\rangle \to s}}{[\text{outer}]\ \langle \text{MOO OOO moo moo}, s\rangle \to s}}{[\text{outer:=[MOO OOM MOO OOO moo moo]}] \vdash \langle \text{OOM MOO OOO moo moo}, s\rangle \to s}}{[\ ] \vdash \langle \text{MOO OOM MOO OOO moo moo}, s\rangle \to s}$$

Toelichting

De eerste `MOO` in het programma begint een lus die gewoon wordt uitgevoerd. Direct voor de uitvoering wordt de inhoud van de lus (in feite de rest van het programma) in het environment geplaatst en de executie gaat verder met de instructie na `MOO`. De volgende `MOO` doet precies hetzelfde (want m_p is nog altijd 1) en de rest van het programma na de tweede `MOO` wordt ook in het environment geplaatst. De executie gaat weer verder.

De eerste `moo` zorgt ervoor dat de binnenste lus lust. Hiervoor wordt de opgeslagen code van de binnenste lus uit het environment gehaald en uitgevoerd. m_p is inmiddels 0 geworden, zodat alle instructies na `MOO` worden overgeslagen tot er een bijbehorende `moo` gevonden wordt, waarna executie wordt hervat. Voor de laatste `moo` gebeurt hetzelfde, maar merk op dat na een `MOO` met m_p = 0 altijd instructies worden overgeslagen, totdat de ***bijbehorende*** `moo` wordt gevonden. Dit betekent dat wij in de state het *nesting level* moeten bijhouden (en dat gebeurt ook in dit voorbeeld).

Instructiegeneratie

De hierboven beschreven natuurlijke semantiek is pas compleet als wij ook de instructie `mOO` behandelen, die een geheugenwaarde omzet in een instructie. Met onze loopdefinities werkt dit precies goed. De natuurlijke semantiek van `mOO` kan in een groep regels gedefinieerd worden. Er is geen regel voor $m_p = 3$, want dat zou een oneindige loop veroorzaken.

$$[\mathrm{mOO}^0] \qquad \frac{\langle \mathrm{moo}, s\rangle \to s'}{\langle \mathrm{mOO}, s\rangle \to s'} \qquad \text{als } m_p = 0$$

$$[\mathrm{mOO}^1] \qquad \frac{\langle \mathrm{mOo}, s\rangle \to s'}{\langle \mathrm{mOO}, s\rangle \to s'} \qquad \text{als } m_p = 1$$

$$[\mathrm{mOO}^2] \qquad \frac{\langle \mathrm{moO}, s\rangle \to s'}{\langle \mathrm{mOO}, s\rangle \to s'} \qquad \text{als } m_p = 2$$

$$[\mathrm{mOO}^4] \qquad \frac{\langle \mathrm{Moo}, s\rangle \to s'}{\langle \mathrm{mOO}, s\rangle \to s'} \qquad \text{als } m_p = 4$$

$$[\mathrm{mOO}^5] \qquad \frac{\langle \mathrm{MOo}, s\rangle \to s'}{\langle \mathrm{mOO}, s\rangle \to s'} \qquad \text{als } m_p = 5$$

$$[\mathrm{mOO}^6] \qquad \frac{\langle \mathrm{MoO}, s\rangle \to s'}{\langle \mathrm{mOO}, s\rangle \to s'} \qquad \text{als } m_p = 6$$

$$[\mathrm{mOO}^7] \qquad \frac{\langle \mathrm{MOO}, s\rangle \to s'}{\langle \mathrm{mOO}, s\rangle \to s'} \qquad \text{als } m_p = 7$$

$$[\mathrm{mOO}^8] \qquad \frac{\langle \mathrm{OOO}, s\rangle \to s'}{\langle \mathrm{mOO}, s\rangle \to s'} \qquad \text{als } m_p = 8$$

$$[\mathrm{mOO}^9] \qquad \frac{\langle \mathrm{MMM}, s\rangle \to s'}{\langle \mathrm{mOO}, s\rangle \to s'} \qquad \text{als } m_p = 9$$

$$[\mathrm{mOO}^{10}] \qquad \frac{\langle \mathrm{OOM}, s\rangle \to s'}{\langle \mathrm{mOO}, s\rangle \to s'} \qquad \text{als } m_p = 10$$

$$[\mathrm{mOO}^{11}] \qquad \frac{\langle \mathrm{oom}, s\rangle \to s'}{\langle \mathrm{mOO}, s\rangle \to s'} \qquad \text{als } m_p = 11$$

Structurele Operationele Semantiek

In tegenstelling tot de natuurlijke semantiek (big-step semantics) beschrijft de structurele operationele semantiek (small-step semantics) hoe het eerste (kleinste) stapje van de uitvoering van een statement plaatsvindt.

Triviale instructies

De instructies die triviaal waren om te beschrijven in de natuurlijke semantiek (alle behalve 0, 3 en 7) zijn dit ook in de structurele operationele semantiek, omdat ze precies uit één stapje bestaan. Het zijn instructies die direct tot een eindtoestand leiden.

Een uitzondering is de compositie van statements, die uit twee delen bestaat.

	[comp[1]]	$\dfrac{\langle S_1, s\rangle \Rightarrow \langle S'_1, s'\rangle}{\langle S_1\ S_2, s\rangle \Rightarrow \langle S'_1\ S_2, s'\rangle}$	Als statement S_1 niet in één stap uitgevoerd kan worden
	[comp[2]]	$\dfrac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1\ S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$	Als statement S_1 in één stap uitgevoerd kan worden
1	[$\text{mOo}^{p=0}$]	$\langle \text{mOo}, s\rangle \Rightarrow s$	als p = 0
2	[moO]	$\langle \text{moO}, s\rangle \Rightarrow s[p \rightarrow p + 1]$	
4	[Moo^{in}]	$\langle \text{Moo}, s\rangle \Rightarrow s[m_p \rightarrow v_0, v \rightarrow \text{tail}(v)]$	als $m_p = 0$
4	[Moo^{out}]	$\langle \text{Moo}, s\rangle \Rightarrow s[v \rightarrow v + m_p]$	als $m_p \neq 0$
5	[MOo]	$\langle \text{MOo}, s\rangle \Rightarrow s[m_p \rightarrow m_p - 1]$	
6	[MoO]	$\langle \text{MoO}, s\rangle \Rightarrow s[m_p \rightarrow m_p + 1]$	
8	[OOO]	$\langle \text{OOO}, s\rangle \Rightarrow s[m_p \rightarrow 0]$	
9	[MMM]	$\langle \text{MMM}, s\rangle \Rightarrow s[r \rightarrow m_p]$	als $r = \varepsilon$
9	[MMM]	$\langle \text{MMM}, s\rangle \Rightarrow s[m_p \rightarrow r, r \rightarrow \varepsilon]$	als $r \neq \varepsilon$
10	[OOM]	$\langle \text{OOM}, s\rangle \Rightarrow s[w \rightarrow w \mathbin{++} m_p]$	
11	[oom]	$\langle \text{oom}, s\rangle \Rightarrow s[m_p \rightarrow v_0, v \rightarrow \text{tail}(v)]$	

Lus-instructies

In de structurele operationele semantiek maken wij, net als bij de natuurlijke semantiek, gebruik van een environment om de bodies van while-loops op te slaan. Door de aanwezigheid van 'halve' while-loops waren wij al genoodzaakt om de natuurlijke semantiek te noteren in een stijl die riekt naar structurele operationele semantiek, en het blijkt dat de denotatie van de laatste niet veel van de eerste verschilt.

Lusexecutie

De regels voor lusexecuties zijn analoog aan de regels in natuurlijke semantiek, met dit verschil dat wij een alleen herschrijven, en niet aangeven hoe de eindtoestand van een lusiteratie eruit ziet (d.w.z. er is géén s').

[MOO^{loop}]	$env \vdash \langle \text{MOO } S, s\rangle \Rightarrow \text{push}(\text{MOO } S, env) \vdash \langle S, s\rangle$	als $m_p \neq 0$
[moo^{loop}]	$env \vdash \langle \text{moo}, s\rangle \rightarrow \text{pop}(env) \vdash \langle S, s\rangle$	met $\text{top}(env) = S$

De functies *push*, *pop* en *top* die toegang bieden tot het environment zijn niet veranderd.

Voorwaarts zoeken

De regels voor voorwaarts door een lus zoeken zijn óók analoog aan de regels in de natuurlijke semantiek. De structurele operationele semantiek is eenvoudiger om te lezen omdat er stapsgewijs door de lus wordt gezocht (iets wat wij in de natuurlijke semantiek alleen met kunst en vliegwerk konden bewerkstelligen).

[MOO$^{\text{begin search}}$]	env ├ <(MOO1 S_1) S_2, s> ⇒ env ├ <MOO2 S_2, s>	als $m_p = 0$
[MOO$^{\text{search}}$]	env ├ <(MOO2 S_1) S_2, s> ⇒ env ├ <MOO2 S_2, s>	$S_1 \neq$ MOO & $S_1 \neq$ moo
[MOO$^{\text{begin nest}}$]	env ├ <(MOO2 MOO) S_2, s> ⇒ env ├ <MOO2 S_2, s[l → l + 1]>	
[MOO$^{\text{end nest}}$]	env ├ <(MOO2 moo) S_2, s> ⇒ env ├ <MOO2 S_2, s[l → l – 1]>	als l > 0
[MOO$^{\text{end search}}$]	env ├ <(MOO2 moo) S_2, s> ⇒ <S_2, s>	als l = 0

Instructiegeneratie

De regels voor instructiegeneratie met `mOO` zijn precies hetzelfde als in de natuurlijke semantiek (waarbij → verandert in ⇒). We laten ze hier weg.

Vergelijking van semantiekdenotatie

Nu wij de natuurlijke en structurele operationele semantiek voor COW gespecificeerd hebben, kunnen we de verschillen beschouwen. De natuurlijke semantiek is geschikt indien men een verband kan leggen tussen de begin- en eindtoestand van de uitvoering van een instructie (zoals dat in de taal While [Nielson92] goed kon). In COW kan dit niet voor lussen, vanwege het bestaan van ‘halve’ while-lussen: er is geen garantie dat op een `MOO` (begin lus) ook een `moo` (einde lus) volgt. In de structurele operationele semantiek kijkt men slechts één (small) step vooruit, waardoor de semantiekregels beter leesbaar zijn.

Turing Completeness

Het rest ons nog te bewijzen dat de taal COW Turing-compleet is. Dit kan bewezen worden op de volgende manieren [Faase]:

1) Laat zien dat er een mapping bestaat van iedere mogelijke Turingmachine naar een programma in COW;
2) Laat zien dat er een COW-programma bestaat dat een universele Turingmachine simuleert;
3) Laat zien dat COW equivalent is met een taal waarbij bekend is dat deze Turing-compleet is;
4) Laat zien dat COW in staat is alle berekenbare functies te berekenen.

Opties 2 en 3 lijken ons zeer complex. Optie 1 is interessant, maar zal resulteren in een groot COW programma dat als bewijs moet dienen. Aangezien het moeilijk is om programma's in COW te lezen, lijkt ons dit geen prettig bewijs. De aanpak is wel duidelijk: als wij kunnen laten zien dat COW in een tabel in het geheugen kan zoeken, daaruit gecodeerde toestandsovergangen van een willekeurige Turingmachine kan halen en uitvoeren, dan hebben wij het leeuwendeel al aangetoond.

Berekenbare functies

Wij vinden dat optie 4 ons het meest doorzichtige bewijs oplevert. Volgens [Sudkamp98] zijn de berekenbare functies:

a) De functies
   - de successorfunctie s: $s(x) = x + 1$
   - de nulfunctie z: $z(x) = 0$
   - de projectiefunctie $p_i^{(n)}$: $p_i^{(n)} (x_1 \ldots x_n) = x_i$, $1 \leq i \leq n$
b) Functies gemaakt door functionele compositie, e.g.
   $f(x_1,\ldots, x_k) = h(g(x_1,\ldots, x_k), \ldots, g(x_1,\ldots, x_k))$
   De berekenbaarheid van f volgt als g en h berekenbaar zijn.
c) Functies gemaakt door primitieve recursie.
   Indien g en h berekenbaar zijn (met n resp. n+2 argumenten), dan is
   $f (x_1,\ldots, x_n, 0) = g (x_1, \ldots, x_n)$
   $f (x_1,\ldots, x_n, y+1) = h (x_1, \ldots, x_n, y, f (x_1, \ldots, x_n, y))$
   ook berekenbaar.
d) De functie f gemaakt door *minimalisatie* is ook berekenbaar:
   $f (x_1,\ldots, x_n) = \mu z [p(x_1,\ldots, x_n, z)]$
   indien p een berekenbaar predikaat met n+1 variabelen is. Minimalisatie levert de kleinste z op waarvoor $p(x_1,\ldots, x_n, z)$ waar (1) is.

Aanpak van het bewijs

Wij moeten nu aantonen dat

a) er een functiemechanisme in COW gemaakt kan worden;
b) s, z en p in COW bestaan;
c) functies in COW samengesteld kunnen worden;
d) een functie zichzelf in COW recursief kan aanroepen;
e) minimalisatie in COW kan worden geïmplementeerd (met recursie).

Indien COW al deze dingen kan, dan is de taal Turing compleet.

Functiemechanisme

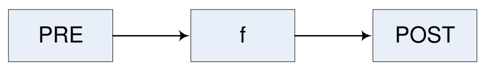
Wij stellen dat

a) als COW in staat is argumenten voor een functie klaar te zetten, ergens in het geheugen (de ‘stack’), en

b) en een stuk code (aangemerkt als functie) deze argumenten kan gebruiken en na uitvoering het eerste argument kan vervangen door het functieresultaat

dan hebben wij een functiemechanisme. Het prettige aan dit mechanisme is dat compositie van functies direct volgt: als de functie *g* in *f o g* resultaat *x* oplevert, dan staat dit resultaat meteen op de plek waar *f* zijn (enige) argument verwacht.

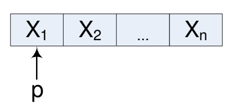
We zullen definiëren dat een functieaanroep bestaat uit een *prelude*, waarin de argumenten klaar worden gezet voor gebruik door een functie *f*, en een *postlude*, die eventueel opruimwerk verricht.



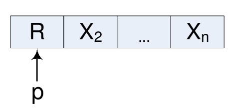
De *prelude* doet het volgende:

- Beweegt de geheugenpointer naar het stuk geheugen waar de functieargumenten zullen worden geplaatst. Dit kan met de instructies `mOo` en `moO`.
- Voor ieder argument x_i:
  - Zet m[p] := x_i. Dit kan met `OOO` en `MoO`.
  - Indien er meer argumenten volgen, verplaatst de prelude de geheugenpointer één positie verder (met `moO`).
- Verplaatst de geheugenpointer terug naar het eerste argument.

De geheugeninhoud vanaf de geheugenpointer *p* ziet er dus na de prelude als volgt uit:



De body van een functie volgt na de prelude, en mag aannemen dat de geheugenpointer naar het eerste argument wijst. De functie weet hoeveel argumenten hij heeft, en verzorgt de code die de argumenten ophaalt. De functie is verantwoordelijk voor het vervangen van zijn eerste argument door zijn functieresultaat. Het resultaat na het uitvoeren van een functie *f* is dus:



Hierbij stelt *R* het functieresultaat voor. Merk op dat de waarden van $x_2 \ldots x_n$ ongedefinieerd zijn. De functie is vrij om ze aan te passen. Het zou wellicht beter zijn als *R* niet een van de functieargumenten zou overschrijven (vgl. het C functie aanroep mechanisme), zodat de prelude dan ruimte voor *R* op de stack zou moeten reserveren, maar deze aanpak is voor ons voldoende.

De *postlude* heeft geen werk te doen. Met dit mechanisme kan de code die volgt op de functie *f* het functieresultaat gebruiken, ook als dit weer een functie *g* is (met één argument). Hoogstens kunnen wij een postlude maken die de gebruikte argumenten (behalve het resultaat) op stack gelijk aan nul maakt, maar dat is overbodig.

Basisfuncties

We tonen aan dat de functies s, z en $p_i^{(n)}$ in COW bestaan door ze te definiëren.

De successorfunctie is de volgende:

s ≡ `MoO`

De nulfunctie is:

z ≡ `OOO`

De projectiefunctie is:

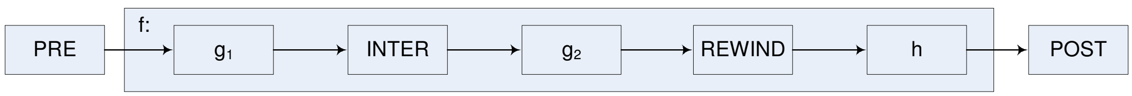
$p_i^{(n)}$ ≡ `moO`$^{i-1}$ `MMM` `mOo`$^{i-1}$ `MMM`

Functiecompositie

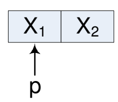
Functiecompositie definiëren wij als:

Laat *f* de compositie van een functie *h*: $\mathrm{N}^n \rightarrow \mathrm{N}$ met de functies $g_1, g_2, ..., g_n$, alle $\mathrm{N}^k \rightarrow \mathrm{N}$. Alle alle g_i en *h* berekenbaar zijn, dan is f ook berekenbaar.

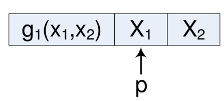
In COW ziet een dergelijke constructie er als volgt uit (voor $h(g_1(x_1,x_2),g_2(x_1,x_2))$):



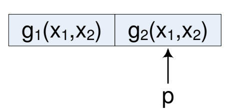
De prelude plaatst nu de argumenten van de functie g_1 op de stack, zodat déze stack ontstaat (voor de voorbeeldfunctie):



De functie g_1 werkt op argumenten x_1 en x_2, en plaatst zijn resultaat op positie *p*. Tussen twee functies g_i en g_j plaatsen wij nu een *interlude,* die de geheugenpointer *p* met 1 verhoogt en de argumenten voor g_j op de stack plaatst. Het resultaat van de interlude na g_1 in dit voorbeeld:



Uitvoering van g_2 levert:



Om *h* uit te kunnen voeren op de resultaten van g_1 en g_2 hebben wij nog het element *rewind* nodig. Deze code brengt de geheugenpointer terug naar het eerste resultaat, d.w.z. de instructie `mOo` wordt *n-1* maal uitgevoerd. Hierna kan *h* direct uitgevoerd worden op de argumenten die op de stack staan. *Rewind* is dus de prelude van *h.*

Uit dit voorbeeld volgt dat men in COW functies kan samenstellen.

Primitieve Recursie

COW beschikt niet over een echt functie *call* mechanisme. Het enige gereedschap dat wij hebben om recursie te implementeren is de while-loop. Wij kunnen dit bewerkstelligen door de recursie bottom-up te evalueren.

Primitieve recursie is gedefinieerd als:

$$f(x_1,\ldots,x_n,0) = g(x_1,\ldots,x_n)$$
$$f(x_1,\ldots,x_n,y+1) = h(x_1,\ldots,x_n,y,f(x_1,\ldots,x_n,y))$$

Als wij nu *eerst* g (x_1, …, x_n) uitrekenen, en de functie *h* toepassen op het resultaat (en een geschikte *y*), en de toepassing van *h* zo vaak herhalen als nodig is, dan bereiken wij het gewenste resultaat. Natuurlijk wordt een recursie altijd zo uitgerekend, maar wij maken de volgorde hier expliciet, zodat wij de recursie kunnen oplossen met een while-loop.

Diagrammen voor deze implementatie worden te complex. Wij illustreren de aanpak door de inhoud van de stack te beschouwen gedurende het verloop van de berekening, waarbij $\underline{x}$ een vector x_1, x_2, …, x_n voorstelt.

$\underline{x}$, y	De prelude van de functie *f* plaatst $\underline{x}$ en *y* op de stack.
$\underline{x}$, y, g($\underline{x}$)	*g($\underline{x}$)* wordt geëvalueerd. Dit is de basis van de recursie. Een postlude zorgt ervoor dat het resultaat van *g($\underline{x}$)* achter de argumenten $\underline{x}$ en *y* komt te staan.
$\underline{x}$, y = 1, g($\underline{x}$), c	Wij stellen nu *c := y,* en *y := 1*. De teller *c* houdt bij hoeveel iteraties van de recursie nog moeten worden doorlopen. Omdat deze teller afloopt (in plaats van oploopt) kunnen wij een while-loop gebruiken.
$\underline{x}$, 1, h($\underline{x}$,1,g($\underline{x}$)) c-1	Wij voeren *h* uit op $\underline{x}$*, y* en *g($\underline{x}$).* Hier is *g($\underline{x}$)* de waarde van *f* voor *($\underline{x}$, y-1).*
$\underline{x}$, 2, h($\underline{x}$,2,h($\underline{x}$,1,g($\underline{x}$))) c-2	Wij voeren *h* uit op $\underline{x}$, *y* en *h($\underline{x}$,2,f)*, waarbij *f* de vorige waarde van de recursie voorstelt.
$\underline{x}$, 3, h($\underline{x}$,3,h($\underline{x}$,2,h($\underline{x}$,1,g($\underline{x}$)))) c-3	Wij voeren *h* uit op $\underline{x}$, *y* en *h($\underline{x}$,3,f)*, waarbij *f* de vorige waarde van de recursie voorstelt.
h($\underline{x}$, y, f($\underline{x}$, y))	Als de teller *c* de waarde 0 bereikt, dan houdt de recursie op. Dit betekent dat in onze implementatie, de whileloop niet meer wordt herhaald. Een postlude zorgt ervoor dat het resultaat *h($\underline{x}$,y,f($\underline{x}$,y))* op de eerste geheugenpositie van de stack komt te staan, zodat het beschikbaar is voor bijvoorbeeld de volgende functie in een compositie.

Om de berekening die in deze stack wordt voorgesteld uit te voeren, moet COW in staat zijn om

- Functieargumenten te kopiëren;
- Functieargumenten te verplaatsen;
- Een while-loop uit te voeren.

In deze operaties voorziet COW (zoals reeds duidelijk is geworden in de besprekingen van het functiemechanisme en functionele compositie), zodat wij van mening zijn dat COW primitieve recursie kan uitvoeren.

Minimalisatie

Minimalisatie maakt ook gebruik van recursie, maar de oplossing is hier eenvoudiger. Wij illustreren minimalisatie weer met een stackrij:

$\underline{x}$	De prelude van de functie *f* plaatst $\underline{x}$ op de stack.
1, $\underline{x}$	De terminatie-conditie van de minimalisatie (is er al een geschikte *z*?) is voorlopig onwaar. Wij representeren dit als 1, want deze waarde wordt gebruikt om te bepalen of de volgende while-loop moeten worde uitgevoerd.
1, 0, $\underline{x}$, 0	Wij beginnen met *z = 0*. Een kopie van *z* staat op ook de tweede stackpositie (en achter $\underline{x}$).
1, 0, p($\underline{x}$, 0) = 0	Het predikaat *p* wordt geëvalueerd met argumenten $\underline{x}$ en *z*. Laat het 0 opleveren.

1, 0, p(x, 0) = 0	Het resultaat wordt gekopieerd naar de eerste stackpositie, en dan geïnverteerd.
1, 1, x, 1	De while-loop herhaalt zich. *z* wordt met 1 verhoogd en klaargezet. Wij hadden de reservekopie van z nodig om de vorige waarde te achterhalen.
1, 1, p(x, 1) = 1	Het predikaat *p* wordt geëvalueerd met argumenten x en *z*. Laat het 1 opleveren.
0, 1, p(x, 1) = 1	Het resultaat wordt gekopieerd naar de eerste stackpositie, en dan geïnverteerd. De while-loop stopt nu, omdat de terminatieconditie (eerste stackpositie) 0 is.
1, …	De huidige waarde van z wordt naar de eerste stackpositie gekopieerd en vormt het resultaat van de minimalisatie.

Met een while-loop kan dus minimalisatie in COW gerealiseerd worden.

Resultaat

Wij hebben laten zien dat er in COW een functiemechanisme bestaat, waarmee de successorfunctie, de nulfunctie, de projectiefunctie, compositie van functies, primitieve recursie en minimalisatie geïmplementeerd kunnen worden. Daaruit kunnen wij concluderen dat COW Turing compleet is.

Referenties

[Faase] Faase, Frans: *Brainf*** is Turing-Complete*
http://home.planet.nl/~faase009/Ha_bf_Turing.html

[Heber] Heber, Sean: COW – *Programming for Bovines*
http://www.bigzaphod.org/cow/

[Nielson92] Nielson, Hanne Riis & Nielson, Flemming: *Semantics with Applications: a Formal Introduction*, John Wiley & Sons, 1992.
http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.pdf

[Sudkamp98] Sudkamp, Thomas: *Languages and Machines*, 2nd edition, Addison Wesley Longman, 1998.