# Een natuurlijke semantiek voor prototype oververing en lexicaal bereik

Kelley van Evert & Tim Steenvoorden

28 mei 2012

# Inhoudsopgave


**1 Inleiding** **1**

**2 Notatie en terminologie** **2**
2.1 Functies . . . 2
2.2 Partiële functies . . . 2
2.3 Eindige functies, eindige verzamelingen . . . 2
2.4 Tupels . . . 3
2.5 Beschouwing semantisch model . . . 3
2.6 Notationele conventies . . . 4

**3 Taal en syntax** **5**
3.1 Voorbeeldprogramma's . . . 5
3.1.1 Lexical scope . . . 6
3.1.2 Prototype overerving . . . 7
3.2 Grammatica . . . 10

**4 Semantisch model** **11**
4.1 Bindingen . . . 11
4.2 Scope en omliggende scopes . . . 12
4.3 Objecten en prototype overerving . . . 13
4.4 Waarden: referenties en primitieven . . . 13
4.4.1 Natuurlijke getallen . . . 14
4.4.2 Functies . . . 14
4.4.3 Objecten . . . 14
4.5 Locaties en geheugen . . . 14

**5 Natuurlijke Semantiek** **15**

**6 Case study: Wiskundige formulering** **18**

# 1 Inleiding

- Motivatie
- JavaScript
- Lexical scope – vrije/gebeonden variabelen
- Objecten en prototype overerving

# 2 Notatie en terminologie

## 2.1 Functies

In dit werkstuk identificeren we een functie met zijn grafiek, d.w.z. een functie $f : X \to Y$ wordt gedefiniëerd door de verzameling paren $(x, y) \in X \times Y$ waarvoor we beweren dat $f(x) = y$. Uiteraard voldoet zo'n verzameling $f \subseteq X \times Y$ aan de voorwaarde dat $\nexists_{x \in X, y_1 \in Y, y_2 \in Y}[(x, y_1) \in f \wedge (x, y_2) \in f \wedge y_1 \neq y_2]$].

## 2.2 Partiële functies

Vrijwel alle functies die we in dit werkstuk behandelen zijn partiële functies. Wanneer een partiële functie $f$ niet gedefiniëerd is op een zeker punt $x$, schrijven we $f(x) = \perp$. Wanneer het omgekeerde het geval is, schrijven we $f(x) \neq \perp$.

Voor een willekeurige term $\phi = \ldots f(x) \ldots$, waarbij $f$ een partiële functie is die niet gedefiniëerd is op punt $x$, geldt dat $\phi = \perp$. Op deze manier hoeven we niet iets omslachtigs te schrijven als: "als $f(x) = \perp$, dan $z = \perp$; anders als $f(x) \neq \perp$, dan $z = \phi$".

## 2.3 Eindige functies, eindige verzamelingen

De reden dat de meeste behandelde functies partiëel zijn is omdat meeste onderdelen van ons semantisch model *eindig* van karakter zijn. Zo zou het ongewoon zijn om in een programmeertaal gebruik te maken van oneindige objecten, of scopes waarin oneindig veel variabelen bestaan.

Wanneer we het over een eindige functie $f$ hebben, bedoelen we daarmee dat het domein van die functie een eindige verzameling is. Dit kan worden uitgedrukt door te zeggen dat er een zeker getal $n \in \mathbb{N}$ bestaat, zó dat $\underline{n}$ (gedefiniëerd als: $\{m \in \mathbb{N} \mid m \leq n\}$) isomorf is aan het domein van $f$.

$$f : X \to Y \text{ is "eindig" } \stackrel{\text{def}}{=} \exists_{n \in \mathbb{N}} \left[ \underline{n} \cong \{x \in X \mid f(x) \neq \perp\} \right]$$

We schrijven $Y^{\hookleftarrow X}$ voor de verzameling functies $\{f : X \to Y \mid f \text{ "eindig"}\}$.

## 2.4 Tupels

We zullen meermaals in ons werkstuk gebruik maken van willekeurig grote, maar altijd eindige, “lijsten” van elementen uit een zekere verzameling: *tupels*. Deze tupels worden gerepresenteerd door eindige (partiële) functies $t : \mathbb{N} \to X$, als X de verzameling element in kwestie is, waaraan nog een paar extra voorwaarden worden gesteld. De verzameling van alle tupels op een zekere verzameling $X$, genoteerd $X_{\langle\rangle}$, is als volgt gedefiniëerd:

$$X_{\langle\rangle} \stackrel{\text{def}}{=} \left\{ t : \mathbb{N} \to X \mid \exists_{N \in \mathbb{N}} \left[ \forall_{n<N} [t(n) \neq \bot] \wedge \forall_{n \geq N} [t(n) = \bot] \right] \right\}$$

We schrijven $\langle\rangle$, maar ook wel $\varnothing$, voor de lege tupel (deze is natuurlijk hetzelfde voor elke waarden verzameling $X$).

We schrijven $\langle x_0, x_1, \ldots, x_{N-1} \rangle$ voor de tupel $t$ waarvoor geldt: $\forall_{n<N}[t(n) = x_n]$, en: $\forall_{n \geq N}[t(n) = \bot]$.

Als $t$ een tupel is $\in X_{\langle\rangle}$, en $x$ een element van $X$, dan schrijven we $t : x$ voor de tupel $t' = t[N \mapsto x]$, waarbij $N = \min\{n \in \mathbb{N} \mid t(n) = \bot\}$.

## 2.5 Beschouwing semantisch model

We definieren in dit werkstuk een natuurlijke semantiek, d.w.z. een ?-ste orde logica, met axioma’s en deductieregels, en een bijbehorende structuur waarin deze zich afspeelt.

Deze structuur, die we ook wel het *semantisch model* zullen noemen, heeft onderstaand opgesomde elementen. Deze worden verderop precies gedefinieerd, onderstaande opsomming geeft slechts een algemeen beeld.

$\mathbb{M}$
: De verzameling mogelijke *geheugens*, welke ook wel als *eindtoestanden* worden geinterpreteerd.

$(Stm \times \mathbb{M} \times \mathbb{L} \times \mathbb{L})$
: De verzameling *toestanden*, ook wel *configuraties*.

$(\longrightarrow)$
: Een tweeplaatsig predikaat welke als eerste argument een element uit de verzameling van toestanden neemt, en als tweede argument een element uit de verzameling van eindtoestanden ($\mathbb{M}\ldots$). De uitspraak $(S, m, \sigma, \tau) \longrightarrow m'$ moet worden geinterpreteerd worden als:

"Het programma $S$, met geheugen $m$, in scope $\sigma$ en met als **this** object $\tau$, resulteert in eindtoestand $m'$, mits $S$ *correct* is".

## 2.6 Notationele conventies

Terwille van elegantie houden we een aantal gebruikelijke notationele conventies aan:

1. Voor elke twee willekeurige tweestemmige predikaten $\mathsf{S}$ en $\mathsf{T}$ (mogelijk ook =), en drie willekeurige elementen $a$, $b$ en $c$, definieren we de afkorting:

   $$a \mathrel{\mathsf{S}} b \mathrel{\mathsf{T}} c \stackrel{\text{def}}{=} a \mathrel{\mathsf{S}} b \wedge b \mathrel{\mathsf{T}} c$$

   in het geval dat deze bewering correct getypeerd is.

2. Op eenzelfde manier definieren we ook de volgende afkorting:

   $$\{a \in A \mid \phi\} \stackrel{\text{def}}{=} \{a \mid a \in A \mid \phi\}$$

[...]

# 3 Taal en syntax

In dit hoofdstuk zullen we de taal presenteren waarvoor we een natuurlijk taal construeren. De taal maakt gebruikt van prototype overerving en lexicaal bereik. Eerst zullen we een aantal voorbeeldprogramma's beschouwen, om zo informeel het karakter van de te formaliseren taal over te brengen. Daarna geven we een rigoreuze definitie met behulp van een BNF grammatica. De structuur van de productieregels van grammatica worden in latere hoofdstukken gebruikt om axioma's en deductieregels op te stellen. Daarmee heeft de grammatica in zekere zin een dubbele functie.

Elk voorbeeldprogramma en zijn toelichtingen worden als volg gepresenteerd:

| | |
|---|---|
| `local` $f$ | *- variabelen moeten worden gedeclareerd* |
| $f$ = `function` $(i)$ `returns` $n$ | |
|     `local` $n$ | |
|     $n = 2 \times (i + 5)$ | |
| | *- $x$ bestaat niet in deze scope* |
| `local` $x$ | *- $x$ is ongedefinieerd (maar wel aanwezig)* |
| $x = f(42)$ | *- $x$ = 89* |

De toelichtingen moeten als informeel commentaar worden beschouwd, waarmee we aan proberen te geven hoe het programma zich gedraagt. Vaak zijn het uitspraken over de toestand waarin het programma zich bevindt, direct na de linker regel te hebben "uitgevoerd".

## 3.1 Voorbeeldprogramma's

Een variabele moet gedeclareerd worden, en pas daarna kan er een waarde aan worden toegekend.

| | |
|---|---|
| | *- $x$ bestaat niet (in deze scope)* |
| `local` $x$ | *- $x$ is ongedefinieerd (maar wel aanwezig)* |
| $x = 5$ | *- $x$ = 5* |

Het concept van declaratie is juist in deze taal, gezien het lexicaal bereik van variabelen, heel belangrijk. Vergelijk het bovenstaande programma fragment bijvoorbeeld met de volgende situatie.

Variabelen hebben geen vaste type. Er zijn drie typen waarden in de taal: getallen, functies en objecten.

```
local x
x = 5                       - de waarde van x is een getal
x = function (){ skip }     - de waarde van x is een functie
x object                    - de waarde van x is een object
```

De taal is object georienteerd.

```
local o
o object
                            - o.f is niet gedefinieerd
o.f = function (){ skip }   - toekenning waarde aan object attribuut
                            - o.f is wel gedefinieerd
o.n = 5
```

Van de drie typen, zijn getallen en functies *primitief*, en objecten *niet primitief*. Primitieve waarde worden zelf gekopieerd (*by-value*), maar van niet-primitieve waarden worden *referenties* gekopieerd (*by-reference*).

```
local x; x = 6
local y; y = x              - x = 6 en y = 6
y = 7                       - x = 6 en y = 7

local p; p.n = 6
local q; q = p              - p en q verwijzen nu naar hetzelfde object
                            - p.n = 6 en q.n = 6
q.n = 7                     - p.n = 7 en q.n = 7
```

### 3.1.1 Lexical scope

Als in een zekere scope een variabele wordt gereferenceerd (nog) niet is gedefinieerd, wordt in omliggende scopes “gezocht” naar een definitie van deze variabele.

```
local x;
local f; f = function (i)
    x = i + 5

f(5)                        - x = 10
```

..maar wanneer deze wel in de huidige scope bestaat, worden omliggende scopes “met rust gelaten”.

```
local x
local f
f = function (i)
    local x
    x = i + 5

f(5)                              - x heeft nog geen waarde
```

Telkens wanneer een functie wordt aangeroepen, wordt een *nieuwe scope* aangemaakt voor lokale variabelen. Variabelen van deze nieuwe scope kunnen later nog gereferenceerd worden, doordat bijvoorbeeld de functie een lokale functie teruggeeft.

```
local f
f = function (n) returns g
    local g
    g = function () returns n
        n = n + 1

local c
c = f(5)                          - c() → 6, 7, 8, ...
```

Voorbeeld 3.1: Een countervoorbeeld

```
local f
f = function(n) returns g
    local g
    g = function() returns n
        n = n + 1

local c
c = f(5)
c()                               # → 6,7,8,...
```

maar dan wat beter geschreven, etc...

### 3.1.2 Prototype overerving

Prototype overerving is een eenvoudige en dynamische variant van object-geörienteerd programmeren. Net als in klassieke object-gebaseerde talen is er sprake van een object waarin *attributen* zijn gedefinieerd. Elk object heeft ook een expliciete *ouder* (of „parent”). Wanneer we binnen een object een attribuut willen evalueren, doen we dit in drie stappen:

1. Bekijk of het attribuut gedefinieerd is in het object zelf. In dat geval weten we de waarde en leveren deze op.
2. Anders zoeken we het attribuut op in de ouder van het object. Ook dan weten we de waarde en leveren deze op.
3. Wanneer ook de ouder het attribuut niet bevat, herhalen we de zoektocht voor alle volgende ouders totdat we het attribuut hebben gevonden.

Ook hier is dus sprake van een boomstructuur.

Het grote verschil tussen object-gebaseerde talen en prototype-gebaseerde talen is dat de tweede geen onderscheid maakt tussen *klassen* en *instanties*. Een prototype heeft beide functies. Neem bijvoorbeeld het prototype `Deur`:

```
local Deur
Deur object
```

We kunnen `Deur` direct als instantie gebruiken door een attribuut te zetten:

```
Deur.open = 1
```

Een `Deur` is standaard open. Maar we kunnen `Deur` ook als een klasse gebruiken door ervan te erven. In prototype-gebaseerde talen heet dit *klonen*:

```
local GeslotenDeur
GeslotenDeur object
GeslotenDeur clones Deur
```

`GeslotenDeur` heeft dan alle attributen van `Deur`:

```
GeslotenDeur.open            # => 1
```

Maar een `GeslotenDeur` moet natuurlijk gesloten zijn. We zetten zijn attribuut `open` op `0`:

```
GeslotenDeur.open = 0
```

Een gewone `Deur` is nog steeds open:

```
Deur.open                    # => 1
```

Attributen worden dus per object bewaard. Door `open` op `0` te zetten in `GeslotenDeur` verandert er niks in `Deur`.

We kunnen net zoveel klonen maken van een object als we willen en net zo diep klonen als we willen. Neem een `GlazenDeur`, dit is natuurlijk ook een `Deur`, maar wel doorzichtig:

```
local GlazenDeur
GlazenDeur object
GlazenDeur clones Deur
GlazenDeur.doorzichtig = 1
```

Een gewone `Deur` heeft het attribuut `doorzichtig` niet, en dus een `GeslotenDeur` ook niet:

```
GeslotenDeur.doorzichtig     # => fout!
```

Maar we kunnen besluiten dat deuren standaard niet doorzichtig zijn:

```
Deur.doorzichtig = 0
```

Zodat ook onze `GeslotenDeur` niet doorzichtig is:

```
GeslotenDeur.doorzichtig     # => 0
```

Maar er geld nog steeds:

```
GlazenDeur.doorzichtig       # => 1
```

We zien dat we met prototypes een zeer flexibele methode hebben om object-geörienteerd te programmeren. Het is niet nodig om de compiler of parser van te voren uit te leggen dat objecten aan bepaalde „blauwdrukken" moeten voldoen. We creëren objecten „on-the-fly", alsmede hun attributen en relaties. Deze methode komt terug in talen als JavaScript, IO en Self.

Natuurlijk is het ook mogelijk om *methoden* te definiëren. Dit zijn functies die gekoppeld zijn aan een specifiek object. Stel dat we een `GeslotenDeur` graag open willen maken. We definiëren:

```
GeslotenDeur.ontsluit = function (poging)
    if poging == THIS.code then
        THIS.open = 1
    else
        THIS.open = 0
```

`THIS` is hier een expliciete verwijzing naar het huidige object. Op dit moment kunnen we `ontsluit` nog niet aanroepen op `GeslotenDeur`:

```
GeslotenDeur.ontsluit(1234)  # => fout!
```

Het attribuut `code` is immers niet gedefinieerd in `GeslotenDeur` noch in zijn prototype `Deur`.

We kunnen natuurlijk een `code` toekennen aan `GeslotenDeur`, maar laten we een specifieke `GeslotenDeur` maken met een `code`:

```
local Kluis
Kluis object
Kluis clones GeslotenDeur
Kluis.code = 4321
```

Wanneer we de methode `ontsluit` aanroepen is deze niet gedefinieerd in `Kluis`, maar wel in zijn prototype `GeslotenDeur`. Die wordt dan uitgevoerd. Een belangrijke observatie is dat `ontsluit` wel wordt aangeroepen op `Kluis`. Dat betekent dat THIS verwijst naar `Kluis` en niet `GeslotenDeur`. Het attribuut `code` wordt dan wel gevonden!

```
Kluis.ontsluit(1234)
Kluis.open                    # => 0
```

Oeps. . . Dat was de verkeerde code, nog een poging:

```
Kluis.ontsluit(4321)
Kluis.open                    # => 1
```

## 3.2 Grammatica

[...en vervolgens helemaal formeel – even uitleggen van BNF etc..]

# 4 Semantisch model

## 4.1 Bindingen

Aan de basis van ons model ligt het begrip van *bindingen*. Een binding si een functie die aan het syntactisch object *Id* een *waarde* toekent. De verzameling van alle bindingen definiëren we als

$$\mathbb{B} \stackrel{\text{def}}{=} \mathbb{V}^{\overset{\leftarrow}{\phantom{x}}\,Id}$$

Wat de waardes $\mathbb{V}$ precies zijn komen we later uitgebreid op terug in **??**. Voor nu is het voldoende om te weten dat in ieder geval de natuurlijke getallen $\mathbb{N}$ deel uitmaken van $\mathbb{V}$

Een binding $b \in \mathbb{B}$ is in eerste instantie leeg. Dit geven we aan met $\emptyset$. We willen natuurlijk *Id*'s kunnen koppelen aan waardes. Hiervoor voeren we een notatie in om $b$ te *updaten*. Om bijvoorbeeld de waarde 5 toe te kennen aan de *Id* $x$ zoals in voorbeeld **??** schrijven we

$$b[x \mapsto 5]$$

zodat wanneer we $x$ „opvragen" in $b$ we weten dat

$$b(x) = 5.$$

Wanneer we meerdere *Id*'s willen koppelen aan waardes, bijvoorbeeld $y$ aan 7 en $z$ aan 9 kan dat met bovenstaande notatie als volgt

$$\big(b[y \mapsto 7]\big)[z \mapsto 9].$$

Wat we afkorten tot

$$b[y \mapsto 7, z \mapsto 9].$$

Bindingen komen veelvuldig terug in ons model. Bij scopes zullen we lokale *variabelen* koppelen aan een waarde. Bij objecten zijn het de *attributen* die een waarde krijgen toegekend. Bij scopes moeten we ook rekening houden met eventuele bindingen in de scope buiten de huidige. Eenzelfde opzet geld voor objecten. Door prototype overerving moeten we op zoek naar een attribuut in het prototype van het huidige object, wanneer het niet gedefinieerd is in het object zelf.

## 4.2 Scope en omliggende scopes

Zoals in **??** informeel is behandeld, zijn scopes goed te representeren met een boomstructuur. Stel we evalueren een variabele $x$ in scope $s$:

```
x                    # De scope waar we ons in bevinden noemen we s.
```

Dan zoeken we eerst $x$ op in de binding $b_s$ behorende bij $s$

$$b_s(x). \tag{4.1}$$

Zoals ook te zien in **??** hebben we twee mogelijkheden:

1. $x$ is gedefinieerd in $b_s$ en we gebruiken de gevonden waarde.
2. $x$ is niet gedefinieerd in $b_s$ en we moeten $x$ opzoeken in de omliggende scope.

We moeten dus niet alleen de bindingen van de scope zelf bijhouden, maar ook een verwijzing naar zijn *omgevende scope*. Een scope $s$ is dan een paar $(b,\pi)$ met $b$ de bindingen ban de eigen scope en $\pi$ een *verwijzing* naar de omgevende scope (of *parent* scope).

We moeten benadrukken dat $\pi$ een *verwijzing* is, en niet een *kopie* van de bindingen van de omgevende scope. Stel dat we het programma in voorbeeld 4.1 uitvoeren. Op het moment dat we $f()$ aanroepen in regel 8 willen we dat $x$ daarna evalueert in 2. Evenzo moet $x$ na regel 9 evalueren in 4. De scope $s_f$ van functie $f$ heeft een eigen binding $b_f$ die leeg is, $x$ is namelijk niet gedeclareerd als `local`. De omgevende scope $\pi_f$ van functie $f$ verwijst naar scope $s$, zodat de variabele $x$ uiteindelijk wel gevonden wordt.

Voorbeeld 4.1: Lexicale scope

```
1 # Buitenste scope genaamd s
2 local x
3 x = 1
4 local f
5 f = function () {            # Introduceert nieuwe scope s_f.
6     x = 2 * x
7 }                            # Einde nieuwe scope
8 f()                          # → x = 2
9 f()                          # → x = 4
```

Stel dat we geen verwijzing in de scope opslaan maar een kopie van de omgevende bindingen. Op het moment dat we $f$ definiëren in regel 5 is scope $s_f$ een paar $(b_f, p_f)$ met $b_f, p_f \in \mathbb{B}$. Net als hierboven zijn de eigen bindingen $b_f$ leeg. De binding $p_f$ bevat een functie onder naam $f$ en de waarde 1 onder naam $x$. Wanneer we $x$ aanpassen door de aanroep in regel 8 wordt dit doorgevoerd in de binding $p_f$ maar, omdat dit een kopie is, niet in de binding $b_s$ van de omgevende scope $s$. We moeten dus wel een verwijzing opslaan willen we het gevraagde gedrag krijgen. Daarnaast wordt het met kopieën erg lastig om een boomstructuur te creëren zodat we een variabele nog hogerop kunnen opzoeken.

Een scope $s$ is dus een element uit de verzameling

$$\mathbb{S} \stackrel{\text{def}}{=} \mathbb{B} \times (\mathbb{L}_s \cup \{\perp\}).$$

Hierbij zijn $\mathbb{B}$ de bindingen zoals besproken in §**??**. $\mathbb{L}_s$ zijn locaties van scopes. Op het begrip locatie komen wij nog terug in §4.5. We moeten er wel rekening mee houden dat er een soort „ultieme" omgevende scope is. Het kan dus zijn dat een scope geen parent heeft. In dat geval zetten we

$$\pi = \perp.$$

We zeggen dat $\pi$ *niet bestaat* of *niks* is. Vandaar dat we het symbool $\perp$ toevoegen aan $\mathbb{L}_s$.

## 4.3 Objecten en prototype overerving

## 4.4 Waarden: referenties en primitieven

[Ze worden op dezelfde manier behandeld: objecten by-reference, dus de references zelf by-value, net als primitieven – vandaar dat ze in dezelfde verzameling waarden zitten.]

### 4.4.1 Natuurlijke getallen

### 4.4.2 Functies

### 4.4.3 Objecten

## 4.5 Locaties en geheugen

## Extra

| | |
|---|---|
| $\mathbb{L} \overset{\text{def}}{=} \{(n,n) \in \mathbb{N}^2\}$ | locaties van scopes en objecten |
| $\mathbb{F} \overset{\text{def}}{=} \mathit{Stm} \times \mathit{Id}_{\langle\rangle} \times (\mathit{Id} \cup \{\underline{\perp}\}) \times \mathbb{L}$ | functies |
| $\mathbb{V} \overset{\text{def}}{=} \mathbb{L} \cup \mathbb{N} \cup \mathbb{F}$ | waarden |
| $\mathbb{B} \overset{\text{def}}{=} \mathbb{V}^{\hookleftarrow \mathit{Id}}$ | binding-verzamelingen |
| $\mathbb{O} \overset{\text{def}}{=} \mathbb{B} \times (\mathbb{L} \cup \{\underline{\perp}\})$ | objecten |
| $\mathbb{S} \overset{\text{def}}{=} \mathbb{B} \times (\mathbb{L} \cup \{\underline{\perp}\})$ | scopes |

# 5 Natuurlijke Semantiek

Deze teksten zijn vooral bedoeld als "tekstvlees" (lorem ipsum's). We zullen axioma's en deductieregels introduceren waarmee we de relatie $(\longrightarrow)$ definiëren, die de volgende signatuur heeft:

$$(\longrightarrow) \subseteq (\mathit{Stm} \times \mathbb{M} \times \mathbb{L} \times \mathbb{L}) \times \mathbb{M}$$

Wanneer we een uitspraak doen van de vorm:

$$\langle S \rangle_{m,\sigma,\tau} \longrightarrow m'$$

..dan bedoelen we daarmee dat:

$$\big((S,m,\sigma,\tau),m'\big) \in (\longrightarrow)$$

Deze uitspraak moet je lezen als: "In de toestand met geheugen $m$, scope $\sigma$ en *this* object $\tau$, termineert het statement $S$, waarbij het resultaat-geheugen $m'$ is."

Een van deze axioma's [object], heeft betrekking tot de productieregel in de grammatica die de `object` "literal" introduceert.

$$\begin{aligned}
&\langle i\ \texttt{object} \rangle_{m,\sigma,\tau} \longrightarrow m'' \\
&\quad \textbf{desda} \quad \textsc{Find}_m(\sigma,i) = \sigma_{\text{def}} \\
&\quad \phantom{\textbf{desda}} \quad m' = m\,[\,\sigma_{\text{def}} \mapsto \big(b_{m(\sigma')}[i \mapsto \omega], p_{m(\sigma')}\big)\,] \\
&\quad \phantom{\textbf{desda}} \quad m'' = m'\,[\,\omega \mapsto (\varnothing, \perp)\,]
\end{aligned}$$

Zoals vele axioma's en deductieregels heeft ook dit axioma een aantal voorwaarden waaraan moet worden voldaan. Deze staan eronder genoteerd, elk op een regel.

Wanneer bij een dergelijke opsomming van voorwaarden een nieuwe variabele wordt geïntroduceerd zoals hierboven, met de volgende vorm: **desda** $\square = \theta \ldots$; dan moet deze gelezen worden als: **desda** $\exists_\theta[\,\square = \theta \ldots\,]$.

$$\begin{aligned}
&\langle i\ \texttt{clones}\ j \rangle_{m,\sigma,\tau} \longrightarrow m' \\
&\quad \textbf{desda} \quad [[\,i\,]]_{m,\sigma,\tau} = \omega_i \in \mathbb{L} \\
&\quad \phantom{\textbf{desda}} \quad [[\,j\,]]_{m,\sigma,\tau} = \omega_j \in \mathbb{L} \\
&\quad \phantom{\textbf{desda}} \quad m' = m\,[\,\omega_i \mapsto \big(b_{m(\omega_i)}, \omega_j\big)\,]
\end{aligned}$$

bla bla

$$\langle \texttt{local}\ i \rangle_{m,\sigma,\tau} \longrightarrow m'$$

$$\textbf{desda} \quad m' = m\,[\,\sigma \mapsto \big(b_{m(\sigma)}[i \mapsto \perp], p_{m(\sigma)}\big)\,]$$

bla bla

$$\langle \texttt{this.}s = e \rangle_{m,\sigma,\tau} \longrightarrow m'$$

$$\begin{aligned} \textbf{desda} \quad & \textsc{Trav}_m(\tau, s) = (\omega, i) \\ & [[\,e\,]]_{m,\sigma,\tau} = v \\ & m' = m\,[\,\omega \mapsto \big(b_{m(\omega)}[i \mapsto v], p_{m(\omega)}\big)\,] \end{aligned}$$

bla bla

$$\langle i = e \rangle_{m,\sigma,\tau} \longrightarrow m'$$

$$\begin{aligned} \textbf{desda} \quad & \sigma_{\text{def}} = \textsc{Find}_m(\sigma, i) \\ & [[\,e\,]]_{m,\sigma,\tau} = v \\ & m' = m\,[\,\sigma_{\text{def}} \mapsto \big(b_{m(\sigma_{\text{def}})}[i \mapsto v], p_{m(\sigma_{\text{def}})}\big)\,] \end{aligned}$$

bla bla

$$\langle i.s = e \rangle_{m,\sigma,\tau} \longrightarrow m'$$

$$\begin{aligned} \textbf{desda} \quad & \sigma_{\text{def}} = \textsc{Find}_m(\sigma, i) \\ & b_{m(\sigma_{\text{def}})}(i) = \omega \in \mathbb{L} \\ & \textsc{Trav}_m(\omega, s) = (\omega', j) \\ & [[\,e\,]]_{m,\sigma,\tau} = v \\ & m' = m\,[\,\omega' \mapsto \big(b_{m(\omega')}[j \mapsto v], p_{m(\omega')}\big)\,] \end{aligned}$$

bla bla

$$\frac{\langle S_f \rangle_{m',\sigma_{f\text{new}},\omega'} \longrightarrow m''}{\langle i.s(e^*) \rangle_{m,\sigma,\tau} \longrightarrow m''}$$

$$\begin{aligned} \textbf{desda} \quad & \sigma_{\text{def}} = \textsc{Find}_m(\sigma, i) \\ & b_{m(\sigma_{\text{def}})}(i) = \omega \in \mathbb{L} \\ & \textsc{Trav}_n(\omega, s) = (\omega', j) \\ & (S_f, I_f, i_f, \sigma_{f\text{def}}) = f = b_{m(\omega')}(j) \\ & \sigma_{f\text{new}} = \textsc{Next}_{\text{scope}}(m) \\ & m' = m\,[\,\sigma_{f\text{new}} \mapsto \big([[\,e^*\,]]^*_{m,\sigma,\tau}(I_f), \sigma_{f\text{def}}\big)\,] \end{aligned}$$

bla bla

$$\langle \texttt{skip} \rangle_{m,\sigma,\tau} \longrightarrow m$$

bla bla

$$\frac{\langle S_1 \rangle_{m,\sigma,\tau} \longrightarrow m' \qquad \langle S_2 \rangle_{m',\sigma,\tau} \longrightarrow m''}{\langle S_1 ; S_2 \rangle_{m,\sigma,\tau} \longrightarrow m''}$$

bla bla

$$\frac{\langle S_1 \rangle_{m,\sigma,\tau} \longrightarrow m'}{\langle \texttt{if}\ (b)\ \texttt{then}\ \{S_1\}\ \texttt{else}\ \{S_2\} \rangle_{m,\sigma,\tau} \longrightarrow m'}$$

$$\textbf{desda} \quad [\![\, b \,]\!]^{\mathrm{B}}_{m,\sigma,\tau} = \mathbf{T}$$

bla bla

$$\frac{\langle S_2 \rangle_{m,\sigma,\tau} \longrightarrow m'}{\langle \texttt{if}\ (b)\ \texttt{then}\ \{S_1\}\ \texttt{else}\ \{S_2\} \rangle_{m,\sigma,\tau} \longrightarrow m'}$$

$$\textbf{desda} \quad [\![\, b \,]\!]^{\mathrm{B}}_{m,\sigma,\tau} = \mathbf{F}$$

bla bla

$$\frac{\langle S_1 \rangle_{m,\sigma,\tau} \longrightarrow m' \qquad \langle \texttt{while}\ (b)\ \texttt{do}\ \{S_1\} \rangle_{m',\sigma,\tau} \longrightarrow m''}{\langle \texttt{while}\ (b)\ \texttt{do}\ \{S_1\} \rangle_{m,\sigma,\tau} \longrightarrow m''}$$

$$\textbf{desda} \quad [\![\, b \,]\!]^{\mathrm{B}}_{m,\sigma,\tau} = \mathbf{T}$$

bla bla

$$\langle \texttt{while}\ (b)\ \texttt{do}\ \{S_1\} \rangle_{m,\sigma,\tau} \longrightarrow m$$

$$\textbf{desda} \quad [\![\, b \,]\!]^{\mathrm{B}}_{m,\sigma,\tau} = \mathbf{F}$$

# 6 Case study: Wiskundige formulering

De vorm van bovenstaand semantisch model geeft een conceptueel sterk beeld van hoe de taal waarschijnlijk geimplementeerd zou worden in een compiler (modulo technische details, etc...). Het is ook vrijwel volgens bovenstaande semantiek dat aan informatica studenten object geörienteerde talen worden uitgelegd (referenties, primitieve waarden, ...). Je kunt echter ook semantisch model maken wat "wiskundiger van aard" is. Daar zal dit hoofdstuk over gaan.

Belangrijke informatie, zoals welk object van welk ander object een prototype is en hoe de scopes elkaar bevatten, maar ook welke informatie binnen een object zit opgeslagen en welke variabelen in een scope zijn gedefiniëerd, worden ditmaal door relaties bevat.

Rest nog identificatie van objecten en scopes, deze zal op eenzelfde manier worden behandeld als eerder de "locaties" binnen het geheugen: ze moeten identiek zijn, maar verder is het van geen belang hoe ze worden gerepresenteerd. We zullen daarom aannemen dat er twee verzamelingen identifiers zijn, $\mathbb{S}$ en $\mathbb{O}$, waarbij bovendien:

1. Voor beide verzamelingen $X \in \{\mathbb{S}, \mathbb{O}\}$, bestaat er een functie $\textsc{Next} : \mathscr{P}(X) \to X$, zdd $\textsc{Next}(Y) \notin Y$ voor alle $Y \subseteq X$.
2. $\mathbb{S} \cap \mathbb{O} = \varnothing$ (Deze eis is niet een technische benodigdheid, maar geeft wel aan dat de semantiek van deze twee soorten identifiers nu eenmaal anders is.)

De prototype hiërarchie, eveneens de scope hiërarchie, worden natuurlijk heel goed weergegeven door partiële ordeningen. Deze twee relaties zullen zullen we als $\sqsubset^p$ en $\sqsubset^s$, respectievelijk, noteren, en aannemen:

1. $\sqsubset^p$ is een (tweestemmige) partiële ordening op $\mathbb{O}$
2. $\sqsubset^s$ is een (tweestemmige) partiële ordening op $\mathbb{S}$

Vervolgens introduceren we de relatie $attr \subseteq \mathbb{O} \times Id \times (\mathbb{O} \cup \mathbb{P})$. De uitspraak "$p[i] = v$" moet worden gelezen als: $(p, i, v) \in attr$. Deze relatie wordt eveneens gebruikt om de "inhoud" van objecten weer te geven, als de graafstructuur tussen objecten. Soortgelijk definiëren we de relatie $def \subseteq \mathbb{S} \times Id \times (\mathbb{O} \cup \mathbb{P})$, waarbij de uitspraak "$\sigma[i] = v$" moet worden gelezen als: $(\sigma, i, v) \in def$.

Ten alle tijde moeten de prototype en scope hiërarchieën, de inhoud van objecten en scopes, en de laatste indentificaties van objecten en scopes worden bijgehouden, en dit vormt dan het “geheugen”, of de “toestand”: $s = (attr, def, \sqsubset^p, \sqsubset^s, p_{next}, \sigma_{next})$.

$$
\begin{aligned}
\langle \texttt{skip}, s, \tau, \sigma \rangle &\longrightarrow s \\
\langle x = e, s, \tau, \sigma \rangle &\longrightarrow s[\ \sigma_d[x] = e\ ] \\
&\quad \textbf{desda}\ \sigma_d = \max\{\sigma' \in \mathbb{S} \mid \sigma'[x] \wedge \sigma \sqsubseteq^s \sigma'\} \\
\langle x\ \texttt{object}, \tau, \sigma \rangle &\longrightarrow s[\ \sigma_d[x] = p\ ][\ p_{next} \mapsto p + 1\ ] \\
&\quad \textbf{desda}\ \sigma_d = \max\{\sigma' \in \mathbb{S} \mid \sigma'[x] \wedge \sigma \sqsubseteq^s \sigma'\} \\
&\quad \textbf{en}\ p = p_{next_s}
\end{aligned}
$$