Een natuurlijke semantiek voor prototype oververing en lexicaal bereik

Kelley van Evert & Tim Steenvoorden

30 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 3
2.4 Functies uitbreiden 3
2.5 Lijsten 4
2.6 Notationele conventies 5

3 Taal en syntaxis **6**

3.1 Voorbeeldprogramma's 6
3.1.1 Basis 7
3.1.2 Lexicaal bereik 8
3.1.3 Prototype overerving en object oriëntatie 9
3.2 Grammatica 12

4 Semantisch model **15**

4.1 Bindingen 15
4.2 Scope en omliggende scopes 15
4.3 Objecten en prototype overerving 17
4.4 Functies 17
4.5 Waarden: referenties en primitieven 17
4.6 Locaties en geheugen 19

5 Natuurlijke Semantiek **21**

5.1 Expressies 21
5.2 Statements 21

6 Case study: Wiskundige formulering **25**

1 Inleiding

In dit werkstuk presenteren we een natuurlijke semantiek die wij ontworpen hebben om de concepten *lexicaal bereik* en *prototype overerving* in *object-geörienteerde* talen te karakteriseren. Daartoe hebben we een minimale taal ontworpen die geïnspireerd is door de bestaande programmeertalen IO en JavaScript.

Lexicaal bereik (ook wel *static scoping* genaamd) en prototype overerving zijn mooie fenomenen. Ze zijn ook de fundamenten van "The World's Most Misunderstood Programming Language": JavaScript. Maar lexicaal bereik ligt men eigenlijk heel natuurlijk: zo redeneren wiskundigen al meer dan honderd jaar met formules waarin variabelen lexicaal bereik hebben. En prototype overerving is slechts een elegant en simpel alternatief op klassieke overerving, wanneer het gaat om object-geörienteerd programmeren.

Het doel van dit werkstuk is daarom een formele betekenis te geven aan deze concepten, maar dan wel zó dat de interpretatie van de formele uitspraken zo natuurlijk mogelijk en conceptueel verantwoord is. De bedoeling is dus dat je de gewoon Nederlandse interpretatie van een willekeurig axioma of deductieregel tegen zou kunnen komen in een college programmeren:

$$\langle\!\langle\, i\ \texttt{object}\,\rangle\!\rangle_{m,\sigma,\tau} \longrightarrow m''$$

$$\begin{aligned} \textbf{desda}\quad & \textsc{Find}_m(\sigma, i) = \sigma_{\mathrm{def}} \\ & m' = m\big[\,\sigma_{\mathrm{def}} \mapsto \big(b_{m(\sigma')}[i \mapsto \omega]\,,\, p_{m(\sigma')}\big)\,\big] \\ & m'' = m'\big[\,\omega \mapsto (\varnothing\,,\,\perp)\,\big] \end{aligned}$$

"Zoals jullie weten, moeten we bij statische scope eerst de definie van de variabele zoeken in de huidige en daarna omliggende scopes. Daarna kan een nieuw object worden gemaakt en in de heap gezet, en een pointer *naar dit object wordt vervolgens in de variabele gestopt..."*

Na het bespreken van een aantal notationele keuzes en terminologie, presenteren we eerst de minimale taal, vervolgens het semantische model en tenslotte de natuurlijke semantiek die de twee voorgaande aan elkaar koppelt. In de case study die erop volgt gaan we uitvoeriger in op ...

2 Notatie en terminologie

In dit hoofdstuk behandelen we zowel een aantal gebruikelijke wiskundige concepten, als een aantal specifieke notaties en begrippen die in dit werkstuk vaak zullen terugkeren. Gezien het aard van het onderwerp zullen we bijvoorbeeld vaak over *eindige* functies en verzamelingen spreken.

2.1 Functies

In dit werkstuk identificeren we een functie met zijn grafiek, dit wil zeggen dat 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

$$\neg\exists_{x\in X,\, y_1\in Y,\, y_2\in Y}\big[\,(x, y_1) \in f \wedge (x, y_2) \in f \wedge y_1 \neq y_2\,\big]$$

De reden voor deze aanpak is zeker niet fundamenteel: het is gewoonweg handig om $\varnothing$ te schrijven voor een nieuwe, "lege", partiële functie.

2.2 Partiële functies

Vrijwel alle functies die we in dit werkstuk behandelen zijn partiële functies. Wanneer een partiële functie $f : X \to Y$ niet gedefiniëerd is op een zeker punt x (dus $\neg\exists_{y\in Y}\big[\,(x, y) \in f\,\big]$) schrijven we $f(x) = \perp$. Wanneer het omgekeerde het geval is, schrijven we $f(x) \neq \perp$, of kortweg $f(x) = y$ voor de gecombineerde uitspraak dat f wél gedefiniëerd is op x én dat $(x, y) \in f$.

Voor een willekeurige term $\phi = \ldots f(x) \ldots$, waarbij f een partiële functie is die niet gedefiniëerd is op punt x, geldt ook dat $\phi = \perp$. Op deze manier is het niet nodig om te schrijven: "als $f(x) = \perp$, dan $z = \perp$; anders als $f(x) \neq \perp$, dan $z = \phi$". Deze "verkorte schrijfwijze" stelt ons in staat om op een elegantie manier functie definities op te schrijven. Een voorbeeld:

$$
\begin{aligned}
f : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
f(0,0) = 1 \\
f(n, m+1) = f(n, m)
\end{aligned}
$$

In dit voorbeeld geldt voor alle $m \in \mathbb{N}$ dat $f(0,m) = 1$, en voor alle $n \in \mathbb{N} \setminus \{0\}$ is $f(n,m)$ niet gedefiniëerd.

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. Functies worden vaak gebruikt om "een verzameling variabelen die een bepaalde waarde bevatten" te representeren, bijvoorbeeld de gedefiniëerde variabelen in een zekere scope. Het zou ongewoon zijn om in een programmeertaal gebruik te maken van scopes waarin oneindig veel variabelen kunnen 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 $\underline{N}$ gelijkmachtig is aan het domein van f.

$$
\begin{aligned}
&f : X \to Y \text{ is "eindig"} \overset{\text{def}}{=} \\
&\qquad \exists_{N \in \mathbb{N}} \big[\text{ er bestaat een bijectie van } \underline{N} \text{ naar } \{x \in X \mid f(x) \neq \bot\} \big]
\end{aligned}
$$

Hierbij is $\underline{N}$ gedefinieerd als de verzameling van de eerste N natuurlijke getallen, ofwel $\{n \in \mathbb{N} \mid n < N\}$.

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

2.4 Functies uitbreiden

Het is vaak handig om een functie op een later tijdstip *uit te breiden*. Hiermee bedoelen we dat de functie ongewijzigd blijft op alle punten $(x, y) \in f$, behalve één specifiek punt x_1 dat we willen koppelen aan y_1 zodat geldt:

$$f(x_1) = y_1$$

Dit geven we aan in met de notatie:

$$f[\, x_1 \mapsto y_1 \,]$$

Wanneer we meerdere aanpassingen willen maken, bijvoorbeeld x_2 koppelen aan y_2 en x_3 aan y_3 kan dat met bovenstaande notatie als volgt

$$\big(f[\, x_2 \mapsto y_2 \,]\big)[\, x_3 \mapsto y_3 \,]$$

... hetgeen we afkorten tot:

$$f[\, x_2 \mapsto y_2 \,, x_3 \mapsto y_3 \,]$$

Heel precies gezegd, als $f : X \to Y$ een functie is, $x_n \in X$ en $y_n \in Y$, dan:

$$f[\, x_n \mapsto y_n \,] \stackrel{\text{def}}{=} f' \textbf{ desda}$$
$$\forall_{x \in X,\, y \in Y}\big[\, (x\,, y) \in f' \Leftrightarrow \big((x\,, y) \in f \wedge x \neq x_n\big) \vee (x\,, y) = (x_n\,, y_n) \,\big]$$

2.5 Lijsten

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

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

We schrijven $\langle\rangle$, maar ook wel $\varnothing$ aangezien het gewoon een lege functie is zoals beschreven in het voorgaande, voor de lege lijst. Deze is natuurlijk altijd hetzelfde, ongeacht welke invulling wordt gekozen voor X.

Als t een zekere lijst is, dan zeggen we dat hij van *grootte* $N = \min\{n \in \mathbb{N} \mid t(n) = \bot\}$ is. We schrijven $\langle x_0\,, x_1\,, \ldots\,, x_{N-1}\rangle$ voor de lijst t van grootte N waarvoor geldt dat $\forall_{n<N}[t(n) = x_n]$. Als t een lijst is uit $X_{\langle\rangle}$, en x een element van X, dan schrijven we

$t : x$, de toevoeging van x aan de lijst t, voor de lijst $t' = t[N \mapsto x]$, waarbij N de grootte van t is.

2.6 Notationele conventies

Terwille van leesbaarheid en elegantie houden we een aantal gebruikelijke notationele conventies aan.

Veel wiskundige formules zijn van de vorm $t_1\ R\ t_2$, waarbij R een zeker predikaat is (mogelijk $=$), en t_1 en t_2 termen. Dit soort formules zullen we wel vaker "samenstellen" tot formules als:

$$6 = 2 \cdot 3 > 2 \geq 42 - 40$$
$$f(x) = y \in Y$$

De intentie is enkel een elegante schrijfwijze te hanteren die makkelijk en intuïtief leest. Als we bovenstaande formules uitschrijven krijgen we:

$$6 = 2 \cdot 3 \wedge 2 \cdot 3 > 2 \wedge 2 \geq 42 - 40$$
$$f(x) = y \wedge y \in Y$$

3 Taal en syntaxis

In dit hoofdstuk presenteren we de taal presenteren waarvoor we een natuurlijk semantiek construeren. De taal maakt gebruikt van prototype overerving en lexicaal bereik. Eerst beschouwen we een aantal voorbeeldprogramma's, om zo informeel de te formaliseren taal te karakteriseren. Daarna geven we een rigoreuze definitie met behulp van een BNF grammatica.

De structuur van de productieregels van deze grammatica worden in latere hoofdstukken gebruikt om axioma's en deductieregels op te stellen. Daarmee heeft de grammatica in zekere zin een dubbele functie.

Het is belangrijk om te vermelden dat het ons hierbij niet gaat om een "goed uitziende" taal te maken, maar enkel om er de essentiële onderdelen in te verwerken die nodig zijn voor ons doeleinde: het formaliseren van lexicaal bereik en prototype overerving met een natuurlijke semantiek. Om diezelfde reden moet de syntaxis van de taal voornamelijk worden beschouwd als wat een mogelijke representatie van een *abstract syntax tree* van een "echte" programmeertaal zou kunnen zijn.

3.1 Voorbeeldprogramma's

Elk voorbeeldprogramma en zijn toelichtingen worden als volg gepresenteerd:

Code fragment 3.1. Het eerste voorbeeldprogramma

```
local f                          — f moet eerst worden gedefiniëerd
f = function (i) returns n
  local n
  n = 2 × (i + 5)
                                 — x bestaat niet in deze scope
local x                          — x heeft nog geen waarde, maar is wel gedefiniëerd
x = f(42)                        — x heeft nu de waarde 94
```

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".

3.1.1 Basis

Declaratie van variabelen

Een variabele moet altijd eerst worden gedeclareerd, alvorens er een waarde aan kan worden toegekend of het op andere manieren kan worden gebruikt. Een programma waarin variabelen worden gereferenceerd die nooit zijn gedefiniëerd is niet valide. In code fragment 3.2 zie je hoe declaratie plaatsvindt.

Code fragment 3.2. Declaratie van variabelen

```
          — x bestaat (nog) niet
local x   — x heeft nog geen waarde, maar is wel gedefiniëerd
x = 5     — x bevat nu de waarde 5
```

Het concept van declaratie is juist in deze taal heel belangrijk, gezien het lexicaal bereik van variabelen. Wat lexicaal bereik precies inhoudt wordt weldra behandeld.

Types

Variabelen kunnen na declaratie waarden aannemen. Onze taal bevat waarden van drie types (natuurlijk getal, functie, object), en het onderscheid tussen deze types wordt op *dynamisch* niveau gemaakt i.p.v op syntactisch niveau. Dat houdt in dat een willekeurige variabele elke willekeurige waarde kan aannemen, van elk willekeurig type. Ook kan het in zijn levensduur waarden van meerder types bevatten. Code fragment 3.3 geeft dit idee weer.

Code fragment 3.3. Waarden van verschillende types

```
local x                 — declaratie zonder type indicatie
x = 5                   — type "natuurlijk getal"
x = function (){...}    — type "functie"
x object                — type "object", deze rare initiälisatie heeft een reden
```

3.1.2 Lexicaal bereik

De *scope* (ook wel *bereik*) van een variabele in een programma, is het deel van het programma waarin zij kan worden gebruikt. En zijn verschillende manieren om dit bereik te definiëren, een daarvan is *lexicaal bereik* (ook wel *lexical* of *static scoping*) er één.

Gegeven een bepaalde definitie van dit bereik, is de vraag: Als ik een variabele naam tegenkom, over welke variabele heb ik het dan? Code fragmenten 3.4 en 3.5 illustreren deze vraag. Met *lexicaal bereik* wordt deze dus ruwweg in de “lexicale omgeving” van de referentie gezocht.

Code fragment 3.4. Zoek de definitie van variabele x

```
local x          — dit is de gezochte definitie van x
x = 42

local f
f = function ( )
  ...x...        — waar is deze x gedefiniëerd?
```

Code fragment 3.5. Zoek de definitie van variabele x

```
local x
x = 42

local f
f = function ( )
  local x        — dit is de gezochte definitie van x
  x = 43
  ...x...        — waar is deze x gedefiniëerd?
```

Als je de code zó indenteert zoals in bovenstaande twee code fragmenten (3.4 en 3.5): bij elke functie definitie wordt een regel geïndenteerd, dan komt de indentatie ongeveer overeen met de boomstructuur van de scopes. Het moge duidelijk zijn dat als men “naar buiten toe” zoekt naar de definities van variabelen, een zekere boomstructuur van toepassing is.

Nieuwe scopes ontstaan in onze taal enkel (en altijd) bij functie applicatie. Een functie op zich is een *primitieve waarde*, wat betekent dat er “niks gebeurt” als een functie wordt gedefiniëerd, net als er niks gebeurt wanneer je een getal aan een variabele toekent. Bij functie applicatie, echter, wordt een nieuwe scope aangemaakt, met als *omliggende scope* de scope waarin de functie origineel is gedefiniëerd. Daarin wordt vervolgens de *body* van de functie uitgevoerd. In code fragment 3.6 wordt het belang van dit proces weergegeven: als nieuwe scopes worden aangemaakt bij de definitie van functies, zou de uitvoer van d( ) 8 zijn i.p.v. 43.

Code fragment 3.6. Het belang van creatie van scopes *bij functie applicatie*

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

local c
c = f(5)
c()                          — de eerste aanroep c( ) levert eerst 6 op...
c()                          — de tweede aanroep c( ) levert eerst 7 op...

d = f(42)                    — d( ): 43, 44, 45, 46, ...
```

3.1.3 Prototype overerving en object oriëntatie

Prototype overerving is een variant van object-geörienteerd programmeren. De kern van object-geörienteerd programmeren is het concept van een *object*, dat ertoe dient een verschijnsel uit de werkelijkheid na te bootsen (een reëel object, een patroon, een abstract idee). Het doel is om meer te kunnen programmeren op een conceptueel niveau. Daarmee wordt bijvoorbeeld zowel creatie als onderhoud van de code makkelijker.

Veel objecten zullen natuurlijk gelijke eigenschappen vertonen, of dezelfde structuur hebben. Verder wilt men concepten als specificering en generalisering toepassen op objecten. Deze problemen kunnen op meerdere manieren worden aangepakt. De bekendste variant is *klasse gebaseerde* object-oriëntatie (ook wel *klassieke object-oriëntatie*) en richt zich op het concept van een *klasse*. Objecten van een bepaalde klasse vertonen de structuur en gedrag van die klasse en heten *instanties*. Van specificering is sprake als een klasse eigenschappen van een andere klasse *overerft*. Klassieke object-oriëntatie vind men in talen als Java en C#.

Een andere aanpak met hetzelfde doel is *prototype gebaseerde* object-oriëntatie. Daarbij wordt geen scheiding gemaakt tussen de concepten klasse, die structuur en gedrag specificeert, en instantie, die enkel deze eigenschappen vertoont. In plaats daarvan wordt gewerkt met een prototype structuur, waarbij elk object naar een bepaald *prototype*-object refereert. Nu zijn objecten zelf de dragers van structuur en gedrag.

Technisch gezien werkt prototype overerving als volgt. Van elk object is een prototype bekend, of het heeft geen prototype. Wanneer men een attribuut opvraagt van een zeker object, kan de op te leveren waarde procedureel als volgt worden opgevat:

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

Het grote verschil tussen object-gebaseerde talen en prototype-gebaseerde talen is dus dat de tweede geen onderscheid maakt tussen klassen en instanties. Een prototype heeft beide functies. Neem bijvoorbeeld het object `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. We kunnen `Deur` ook als een prototype gebruiken. In prototype-gebaseerde talen heet dit *klonen*:

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

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

```
GeslotenDeur.open  -- waarde → 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  — waarde → 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  — waarde → 0
```

Maar er geld nog steeds:

```
GlazenDeur.doorzichtig  — waarde → 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 functie attributen gekoppeld 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                — waarde → 0
```

Helaas was dat de verkeerde code, we proberen het nog een keer:

```
Kluis.ontsluit(1234)
Kluis.open                — waarde → 1
```

3.2 Grammatica

Nu volgt een formele definitie van de syntaxis van de taal, aan de hand van een BNF grammatica. Nummers zijn als volgt gedefiniëerd:

$$\textit{Number} ::= (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^{+}$$

Eigenlijk gebruiken we geen strikte BNF, in deze specifieke gevallen, maar een hele simpele variant, zoals E-BNF, die ook reguliere expressies toelaat. Bovenstaand voorbeeld maakt dit duidelijk. Voorbeelden van elementen uit *Number* zijn “0”, “1”, “235783” en “0003”. Voorbeelden van elementen die niet in *Number* zitten zijn “”, “-6”, “4.2”.

Identifiers, die gebruikt worden als namen voor variabelen en attributen, zijn op eenzelfde manier als volgt gedefiniëerd:

$$\mathit{Identifier} ::= (a \mid b \mid c \mid \cdots \mid A \mid B \mid C \mid \ldots)^{+}$$

Hierbij moet men zich voorstellen dat alle letters uit het alfabet in de grammaticaregel staan op de voor de hand liggende manier.

Het is soms ook nodig om meerdere komma-gescheiden namen te gebruiken, of een mogelijk lege lijst, zoals bij functie definities. Vandaar de volgende twee productieregels:

$$\begin{aligned} \mathit{Identifiers} &::= \mathit{Identifier} \mid \mathit{Identifiers}\,,\mathit{Identifier} \\ \mathit{MaybeIdentifiers} &::= \varepsilon \mid \mathit{Identifiers} \end{aligned}$$

Een *pad* is een opeenvolging van identifiers gescheiden door punten en wordt gebruikt om ook naar attributen van objecten te kunnen refereren:

$$\mathit{Path} ::= \mathit{Identifier} \mid \mathit{Identifier}.\mathit{Path}$$

Expressies, die ofwel primitieve waarden (getallen en functies), ofwel objecten kunnen weergeven, en *boolse expressies*, die gebruikt worden voor loops en conditionele executie, definiëren we als volgt:

$$\begin{aligned} \mathit{Expression} &::= \mathit{Number} \mid \mathit{Path} \mid \varnothing \mid \mathit{Expression}\ (+ \mid - \mid \times \mid /)\ \mathit{Expression} \\ &\quad \mid \mathtt{function}\,\mathtt{(}\mathit{MaybeIdentifiers}\mathtt{)}\ [\ \mathtt{returns}\ \mathit{Identifier}\ ]\ \{\ \mathit{Statement}\ \} \\ \mathit{Expressions} &::= \mathit{Expression} \mid \mathit{Expressions}\,,\mathit{Expression} \\ \mathit{MaybeExpressions} &::= \varepsilon \mid \mathit{Expressions} \\ \mathit{BooleanExpression} &::= \mathtt{true} \mid \mathtt{false} \\ &\quad \mid \mathit{BooleanExpression}\ (\mathtt{and} \mid \mathtt{or})\ \mathit{BooleanExpression} \\ &\quad \mid \mathtt{not}\ \mathit{BooleanExpression} \\ &\quad \mid \mathit{Expression}\ (= \mid < \mid \leq \mid > \mid \geq)\ \mathit{Expression} \end{aligned}$$

De kern van de hele grammatica draait om de volgende productieregel voor *statements*. Een statement is een programma van goede vorm. Het betekent niet noodzakelijk dat het programma *valide* is, maar alle valide programma's zitten wel in *Statement*. (Vanwege de focus van dit werkstuk definiëren we niet precies wanneer een programma valide is en wanneer niet.)

$$
\begin{aligned}
\mathit{Statement} ::= {} & \mathtt{\textbf{skip}} \\
& \mid \mathit{Statement} ; \mathit{Statement} \\
& \mid \mathtt{\textbf{if}}\ \mathit{BooleanExpression}\ \mathtt{\textbf{then}}\ \mathit{Statement}\ \mathtt{\textbf{else}}\ \mathit{Statement} \\
& \mid \mathtt{\textbf{while}}\ \mathit{BooleanExpression}\ \mathtt{\textbf{do}}\ \mathit{Statement} \\
& \mid \mathtt{\textbf{local}}\ \mathit{Identifier} \\
& \mid \mathit{Identifier}\ \mathtt{\textbf{object}} \\
& \mid \mathit{Identifier}\ \mathtt{\textbf{clones}}\ \mathit{Identifier} \\
& \mid \mathit{Path} = \mathit{Expression} \\
& \mid [\mathit{Path} =]\ \mathit{Identifier}\,\mathtt{\textbf{(}}\mathit{MaybeExpressions}\mathtt{\textbf{)}}
\end{aligned}
$$

Merk op dat in delen van deze productieregel *Identifier*'s staan waar men misschien een *Path* had verwacht. Zo zou het wenselijk lijken om bijvoorbeeld "`a.b` **`clones`** `c.d`" als een programma van goede vorm te beschouwen. Er zijn twee redenen waarom we dit echter niet hebben gedaan. Ten eerste worden de axioma's en deductieregels ingewikkelder en daarmee minder elegant, of er zijn er meer nodig. Maar belangrijker nog, is het niet essentiëel voor de taal. Voor elk programma zoals "`a.b` **`clones`** `c.d`", bestaat er een equivalent programma zónder zulke paden:

```
a.b clones c.d
```

```
local x
x = a.b

local y
y = c.d

x clones y
```

Hierin moeten x en y "vers" gekozen worden.

4 Semantisch model

4.1 Bindingen

Aan de basis van ons model ligt het concept van een *binding*. Een binding is een toekenning van een *waarde* aan een variabele (een element uit de syntactische verzameling *Identifier*). Bindingen zijn bijvoorbeeld van belang om de gedefiniëerd variabelen binnen een scope vast te leggen, of de attributen van een bepaald object. Een *groep bindingen* is een eindige functie $b : \mathit{Identifier} \rightarrow \mathbb{V}$. De verzameling van alle groepen van bindingen definiëren we dus als

$$\mathbb{B} \stackrel{\text{def}}{=} \mathbb{V}^{\mathit{Identifier}}$$

We komen later terug op wat de waarden $\mathbb{V}$ precies zijn in §4.5. Voor nu is het voldoende om te weten dat in ieder geval de natuurlijke getallen $\mathbb{N}$ deel uitmaken van $\mathbb{V}$.

Bindingen komen veelvuldig terug in ons model. In scopes worden *variabelen* gedeclareerd en aan waarden gekoppeld. Bij objecten zijn het de *attributen* die waarden krijgen toegekend.

4.2 Scope en omliggende scopes

In sectie 3.1 is informeel gebleken dat scopes conceptueel goed te zien zijn als een boomstructuur. Stel we evalueren een variabele `x` in scope s:

```
x — We bevinden ons in een zekere scope s.
```

dan kunnen we dit als volgt uitleggen. Eerst zoeken we `x` op in de bindingen groep b_s, behorende bij scope s.

$$b_s(x).$$

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 definiëren we dus als een paar (b, π), met in b de bindingen en π een *verwijzing* naar de omgevende scope (ook wel *parent*, of *outer* scope).

We moeten benadrukken dat π een *verwijzing* is, en niet een *kopie* van de bindingen groep van de omgevende scope. Stel dat we het programma in code fragment 4.2 uitvoeren. Op het moment dat we $f()$ aanroepen in regel 7 willen we dat x daarna evalueert naar de waarde 2. Evenzo moet x na regel 8 evalueren naar de waarde 4. De scope s_f van functie f heeft een eigen binding b_f die gedurende de executie van het programma leeg is, x is namelijk niet gedeclareerd als een **local** variabele. De omgevende scope π_f van functie f verwijst naar scope s, zodat de variabele x uiteindelijk wel gevonden wordt.

Code fragment 4.2. Lexicale scope: opslaan en vinden van variabelen

```
1  local x
2  x = 1
3  local f
4  f = function () — Introductie nieuwe scope
5    x = 2 × x
6                  — Einde nieuwe scope
7  f()             — x = 2
8  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 4 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 7 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 §4.6. 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

$$\pi = \doteq.$$

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

4.3 Objecten en prototype overerving

In §**??** hebben we een beeld gekregen van prototype overerving. Net als scopes en omgevende scopes, blijken objecten en prototypen te modelleren met een boomstructuur. Geheel in lijn met scopes is een object een paar met daarin zijn eigen bindingen b en een verwijzing naar zijn prototype π. Natuurlijk kan een object ook geen prototype hebben. Dit geven we weer aan met $\doteq$. Een object o is dan een element uit

$$\mathbb{O} \stackrel{\text{def}}{=} \mathbb{B} \times (\mathbb{L} \cup \{\doteq\}).$$

Hierbij zijn $\mathbb{B}$ weer de bindingen uit §**??** en $\mathbb{L}_o$ zijn locaties van objecten. We maken dus een strikte scheiding tussen locaties van scopes en locaties van objecten.

4.4 Functies

4.5 Waarden: referenties en primitieven

In voorgaande paragrafen spraken we telkens over waardes $\mathbb{V}$ en locaties $\mathbb{L}$ ($\mathbb{L}_s$ voor scopes $\mathbb{L}_o$ voor objecten). We hebben de exacte definities hiervan in het midden gelaten. Een waarde $v \in \mathbb{V}$ is iets wat we toekennen aan een *Identifier* met behulp van een binding. Bindingen waren immers functies van *Identifier* naar waardes. In de voorbeelden van §3.1 zijn we verschillende waardes tegengekomen:

```
local n
n = 5                        # Natuurlijke getallen
local f
f = function (x) returns y   # Functies
    y = 2 * x
```

```
local A
A object                      # Objecten
```

In onze taal komen dus drie typen waardes voor. Er is echter een verschil in de manier waarop wij ze behandelen. Wat gebeurt er wanneer we bovenstaande waardes opnieuw toekennen aan andere variabelen? Allereerst een natuurlijk getal, deze is het eenvoudigst:

```
local m
m = n
```

Op dit moment hebben zowel `m` als `n` waarde `5`. Wanneer we aan `m` een andere waarde toekennen, gebeurt er niets met `n`. Omgekeerd geld hetzelfde. We kunnen dus simpelweg de waarde van `n` *kopiëren* en in `m` stoppen. Onze waardes $\mathbb{V}$ bevatten dus sowieso $\mathbb{N}$:

$$\mathbb{V} \stackrel{\text{def}}{=} \mathbb{N}$$

Voor functies geld eigenlijk hetzelfde:

```
local g
g = f
```

`g` heeft nu als waarde een kopie van de functie in variabele `f`. Beiden kunnen we aanroepen:

```
f(4)                          # => 8
g(6)                          # => 12
```

Dus $\mathbb{V}$ bevat ook functies:

$$\mathbb{V} \stackrel{\text{def}}{=} \mathbb{N} \cup \mathbb{V}$$

Helaas gaat dit *niet* op voor objecten. Stel dat `A` een attribuut `x` heeft

```
A.x = 7
```

en we koppelen een nieuwe variabele B aan het object in A:

```
local B
B = A
```

Nu heeft `B` sowieso dezelfde attributen als `A`:

```
B.x                           # => 7
```

Maar als we `x` in `B` ophogen

```
B.x = 8
```

moet dit in `A` natuurlijk ook gebeuren:

```
A.x                           # => 8
```

Wanneer we zomaar een kopie van de waarde in `A` aan `B` toekennen, zou het attribuut `A.x` niet gewijzigd worden. Een oplossing is om in plaats van een kopie van een object een *referentie* naar een object aan een variabele te koppelen. Dit zijn dus niet elementen uit $\mathbb{O}$ zelf, maar uit $\mathbb{L}_o$. Dat betekend dat objecten niet op dezelfde manier worden behandeld als getallen en functies, maar de *locaties* van objecten wel. Voor de waardes krijgen we dus

$$\mathbb{V} \stackrel{\text{def}}{=} \mathbb{L}_o \cup \mathbb{N} \cup \mathbb{V}.$$

Primitieve waardes in onze taal koppelen we *by-value*. Dit zijn natuurlijke getallen en functies. Objecten daarentegen worden *by-reference* gekoppeld. Deze referenties zijn elementen uit $\mathbb{L}_o$. Achter de schermen worden deze gebruikt om objecten door te geven en worden op hun beurt wel *by-reference* gekoppeld.

4.6 Locaties en geheugen

Uit de redenaties over referenties volgt dat we een plaats moeten hebben om alle objecten op te slaan. Dit noemen we het *geheugen voor objecten m_o*. Voor een gegeven locatie $\omega \in \mathbb{L}_o$ kunnen we het bijbehorende object in een geheugen m_o opzoeken met

$$m_o(\omega).$$

Een geheugen is dus een functie van locaties naar objecten. De verzameling van alle geheugens definiëren we als

$$\mathbb{M}_o \stackrel{\text{def}}{=} \mathbb{O}^{\mathbb{L}_o}.$$

Dan is m_o een element hier uit.

Extra

$\mathbb{L} \stackrel{\text{def}}{=} \{(n, n) \in \mathbb{N}^2\}$	locaties van scopes en objecten
$\mathbb{F} \stackrel{\text{def}}{=} \mathit{Statement} \times \mathit{Identifier}_{\langle\rangle} \times (\mathit{Identifier} \cup \{\doteq\}) \times \mathbb{L}$	functies
$\mathbb{V} \stackrel{\text{def}}{=} \mathbb{L} \cup \mathbb{N} \cup \mathbb{F}$	waarden
$\mathbb{B} \stackrel{\text{def}}{=}$ "eindige" $\mathbb{V}^{\mathit{Identifier}}$	binding-verzamelingen
$\mathbb{O} \stackrel{\text{def}}{=} \mathbb{B} \times (\mathbb{L} \cup \{\doteq\})$	objecten
$\mathbb{S} \stackrel{\text{def}}{=} \mathbb{B} \times (\mathbb{L} \cup \{\doteq\})$	scopes

5 Natuurlijke Semantiek

5.1 Expressies

5.2 Statements

Laten we beginnen met de simpelste constructie in onze taal, het lege statement **skip**. Deze heeft de vorm van een axioma.

$$\langle\!\langle\, \texttt{skip} \,\rangle\!\rangle_{m_\mathrm{s}, m_\mathrm{o}, \sigma, \tau} \longrightarrow (m_\mathrm{s}, m_\mathrm{o})$$

Zoals we kunnen zien zijn onze uitspraken van de vorm

$$\langle\!\langle\, S \,\rangle\!\rangle_{m_\mathrm{s}, m_\mathrm{o}, \sigma, \tau} \longrightarrow (m'_\mathrm{s}, m'_\mathrm{o}).$$

Hiermee bedoelen we dat

$$\big(\, (S, m_\mathrm{s}, m_\mathrm{o}, \sigma, \tau), (m'_\mathrm{s}, m'_\mathrm{o}) \,\big) \in (\longrightarrow),$$

waarbij $\longrightarrow$ de volgende signatuur heeft

$$(\longrightarrow) \subseteq (\mathit{Statement} \times \mathbb{M}_\mathrm{s} \times \mathbb{M}_\mathrm{o} \times \mathbb{L}_\mathrm{s} \times \mathbb{L}_\mathrm{o}) \times (\mathbb{M}_\mathrm{s} \times \mathbb{M}_\mathrm{o}).$$

Deze transitie werkt op een statement $S \in \mathit{Statement}$ in een toestand $(m_\mathrm{s}, m_\mathrm{o}) \in (\mathbb{M}_\mathrm{s}, \mathbb{M}_\mathrm{o})$ met als extra informatie de locatie van de huidige scope $\sigma \in \mathbb{L}_\mathrm{s}$ en de locatie van het huidige **this**-object $\tau \in \mathbb{M}_\mathrm{o}$. Het resultaat is een nieuwe toestand in de vorm van de twee geheugens $(m'_\mathrm{s}, m'_\mathrm{o}) \in (\mathbb{M}_\mathrm{s}, \mathbb{M}_\mathrm{o})$. **skip** verandert niets aan de toestand zodat $(m'_\mathrm{s}, m'_\mathrm{o}) = (m_\mathrm{s}, m_\mathrm{o})$.

Voor het samenstellen van statements hebben we een regel nodig.

$$\frac{\langle\!\langle\, S_1 \,\rangle\!\rangle_{m_\mathrm{s}, m_\mathrm{o}, \sigma, \tau} \longrightarrow (m'_\mathrm{s}, m'_\mathrm{o}) \qquad \langle\!\langle\, S_2 \,\rangle\!\rangle_{m'_\mathrm{s}, m'_\mathrm{o}, \sigma, \tau} \longrightarrow (m''_\mathrm{s}, m''_\mathrm{o})}{\langle\!\langle\, S_1; S_2 \,\rangle\!\rangle_{m_\mathrm{s}, m_\mathrm{o}, \sigma, \tau} \longrightarrow (m''_\mathrm{s}, m''_\mathrm{o})}$$

In dit geval geven we aan dat, wanneer we een een compositie hebben van de statements S_1 en S_2, we eerst S_1 evaluerenen daarna S_2. Tijdens dit proces ontstaan nieuwe toestanden, waar we natuurlijk rekening mee moeten houden. De geheugens worden dan ook netjes doorgesluisd.

De controlestructuur `if` heeft twee regels, een voor het geval dat de *BooleanExpression* evalueert in **T** en een voor het geval dat deze evalueert in **F**.

$$\frac{\langle\!\langle\, S_1 \,\rangle\!\rangle_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau} \longrightarrow (m'_{\mathrm{s}}, m'_{\mathrm{o}})}{\langle\!\langle\, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \,\rangle\!\rangle_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau} \longrightarrow (m'_{\mathrm{s}}, m'_{\mathrm{o}})}$$

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

bla bla

$$\frac{\langle\!\langle\, S_2 \,\rangle\!\rangle_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau} \longrightarrow (m'_{\mathrm{s}}, m'_{\mathrm{o}})}{\langle\!\langle\, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \,\rangle\!\rangle_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau} \longrightarrow (m'_{\mathrm{s}}, m'_{\mathrm{o}})}$$

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

Regels

$$\langle\!\langle\, i\ \texttt{clones}\ j \,\rangle\!\rangle_{m,\sigma,\tau} \longrightarrow m'$$

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

bla bla

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

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

bla bla

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

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

bla bla

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

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

bla bla

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

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

bla bla

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

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

bla bla

bla bla

bla bla bla bla

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

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

bla bla

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

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

Deze uitspraak moet je lezen als: "In de toestand met geheugen m, scope σ en *this* object τ, 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.

$$\left\langle\!\left\langle\, i\ \texttt{object}\,\right\rangle\!\right\rangle_{m,\sigma,\tau} \longrightarrow m''$$

$$\begin{array}{ll} \textbf{desda} & \textsc{Find}_m(\sigma, i) = \sigma_{\text{def}} \\ & m' = m\big[\,\sigma_{\text{def}} \mapsto \big(b_{m(\sigma')}[i \mapsto \omega]\,,\, p_{m(\sigma')}\big)\,\big] \\ & m'' = m'\big[\,\omega \mapsto \big(\varnothing\,,\, \perp\big)\,\big] \end{array}$$

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** $\Box = \theta \ldots$; dan moet deze gelezen worden als: **desda** $\exists_\theta\big[\,\Box = \theta \ldots\,\big]$.

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 $\textsc{Next} : \mathcal{P}(X) \to X$, 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": $s = (attr, def, \sqsubset^p, \sqsubset^s, p_{next}, \sigma_{next})$.

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