Een natuurlijke semantiek voor prototype oververing en lexicaal bereik

Kelley van Evert & Tim Steenvoorden

30 mei 2012

Inhoudsopgave

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 JavaScript en IO. JavaScript is een dynamische, prototype-gebaseerde taal die veelvuldig wordt gebruikt bij het ontwikkelen van internet-toepassingen. Een opvallende functie van JavaScript is het gebruik van lexicaal bereik. IO is een onderzoekstaal door Steve Dekorte. Het belangrijkste kenmerk van deze taal is het prototype-gebaseerde object model.

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 dat men de gewoon Nederlandse interpretatie van een willekeurig axioma of deductieregel tegen zou kunnen komen in een college programmeren:

$$\left\langle\!\left\langle\, i\ \texttt{object} \,\right\rangle\!\right\rangle_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau} \longrightarrow (m'_{\mathrm{s}}, m'_{\mathrm{o}})$$

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

"Zoals jullie weten, moeten we bij statische scope eerst de definitie van de variabele zoeken in de huidige en daarna omliggende bereiken. Daarna maken we ruimte vrij in het geheugen en kan een nieuw object worden gemaakt. Een verwijzing 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 proberen we een andere wiskundige aanpak te belichten om de semantiek te beschrijven.

1 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.

1.1 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

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

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

1.2 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 $\neg\exists_{y \in Y} \left[\, (x, y) \in f \,\right]$) 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.

1.3 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.

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

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 $\{f : X \to Y \mid f \text{ "eindig"}\}$.

1.4 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

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

...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:

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

1.5 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:

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

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* $N = \min\{n \in \mathbb{N} \mid t(n) = \perp\}$ 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.

1.6 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:

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

2 Taal en syntaxis

In dit hoofdstuk presenteren we de taal 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 rigoureuze 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 hierbij niet gaat om een taal te maken die er "mooi" uit ziet. Het doel is om de essentiële onderdelen te verwerken die nodig zijn om lexicaal bereik en prototype overerving te formaliseren met een natuurlijke semantiek. Om dezelfde reden moet de syntaxis van de taal worden beschouwd als een mogelijke representatie van een *abstract syntax tree* van een "echte" programmeertaal. We zullen dan ook, waar mogelijk, puntkomma's en haakjes weglaten. Het gebruik van regeleindes en inspringen van blokken geeft, naar ons idee, binder belemmering bij het lezen van een programma.

2.1 Voorbeeldprogramma's

Elk voorbeeldprogramma en zijn toelichtingen worden als volg gepresenteerd:

Code fragment 2.1. 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".

2.1.1 Basis

Declaratie van variabelen

Een variabele moet altijd eerst worden gedeclareerd. Daarna kan er een waarde aan worden toegekend of kan het op andere manieren worden gebruikt. Een programma waarin variabelen worden gebruikt die nooit zijn gedefiniëerd is niet valide. In code fragment 2.2 staat een voorbeeld van declaratie.

Code fragment 2.2. 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

Het onderscheid tussen deze types wordt op *dynamisch* niveau gemaakt in plaats van 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 2.3 geeft dit weer.

Code fragment 2.3. Waarden van verschillende types

```
local x                  — declaratie zonder type indicatie
x = 5                    — type “natuurlijk getal”
x = function ( ){. . . } — type “functie”
x object                 — type “object”, aan x kunnen nu attributen worden toegevoegd
```

2.1.2 Lexicaal bereik

Het *bereik* (ook wel *scope*) van een variabele, is dat deel van het programma waarin zij zichtbaar is. Er zijn verschillende manieren om dit bereik te definiëren. Een daarvan is *lexicaal bereik* (ook wel *lexical* of *static scoping*) dat expliciet gebruikt wordt door JavaScript. De vraag die we ons stellen is: "Als ik de naam van een variabele tegen kom, over welke variabele heb ik het dan?" Code fragmenten 2.4 en 2.5 illustreren deze vraag. We zien dat het sleutelwoord `local` hier een cruciale rol in speelt. De plaats waar een variabele gedeclareerd is, geeft zijn bereik aan. Wanneer een variabele niet in het lokale bereik is gedeclareerd, zoeken we die op in het *omliggende bereik*: het bereik dat lexicaal gezien om het locale bereik heen ligt. Door het nesten van bereiken ontstaat een *boomstructuur*. De niveaus van inspringing in onderstaande voorbeelden komt overeen met de boomstructuur van de bereiken.

Code fragment 2.4. 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 2.5. 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?
```

Een nieuw bereik ontstaat in onze taal enkel 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 nieuw bereik aangemaakt, met als omliggend bereik het bereik waarin de functie was gedefiniëerd. Daarin wordt vervolgens de *body* van de functie uitgevoerd. In code fragment 2.6 wordt het belang van dit proces weergegeven: als nieuwe bereiken worden aangemaakt bij de definitie van functies, zou de uitvoer van `d( )` 8 zijn in plaats van 43.

Code fragment 2.6. Het belang van creatie van bereiken 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 daarna 7 op...

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

2.1.3 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 declareren eerst een locale variabele die we vervolgens initialiseren als een object. Vanaf nu kunnen we `Deur` 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
```

2.2 Formele definitie

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

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

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:

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

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.)

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

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 niet hebben gedaan. Allereerst worden de axioma's en deductieregels ingewikkelder en daarmee minder elegant, of er zijn er meer nodig. Maar belangrijker nog: het is niet essentieel 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.

3 Semantisch model

3.1 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 bereik vast te leggen, of de attributen van een bepaald object. Een *groep bindingen* is een eindige functie $b : \mathit{Identifier} \to \mathbb{V}$. De verzameling van alle groepen van bindingen definiëren we dus als

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

We komen later terug op wat de waarden $\mathbb{V}$ precies zijn in §3.5. 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 bereiken worden *variabelen* gedeclareerd en aan waarden gekoppeld. Bij objecten zijn het de *attributen* die waarden krijgen toegekend.

3.2 Bereik en omliggende bereiken

In sectie 2.1 is informeel gebleken dat bereiken conceptueel goed te zien zijn als een boomstructuur. Stel we evalueren een variabele `x` in bereik s:

```
x  — We bevinden ons in een zeker bereik s.
```

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

$$b_s(x).$$

Vervolgens moet er gevalsonderscheiding worden gedaan voor de volgende situaties:

1. `x` is gedefiniëerd in b_s, dus gebruiken we de gevonden waarde.
2. `x` is niet gedefiniëerd in b_s, dus moeten we `x` opzoeken in de omliggend bereik.

Hieruit blijkt dat we voor een willekeurig bereik niet alleen zijn eigen bindingen moeten bijhouden, maar ook een verwijzing naar zijn *omliggend bereik*. Een bereik s definiëren we daarom als een paar (b, π), met b de bindingen en π een *verwijzing* naar de omliggend bereik (ook wel *parent*).

We moeten benadrukken dat π een *verwijzing* is, en niet een *kopie* van de bindingen groep van de omliggend bereik. Stel dat we het programma in code fragment 3.2 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. Het bereik 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 omliggend bereik π_f van functie f verwijst naar bereik s, zodat de variabele x uiteindelijk wel gevonden wordt.

Code fragment 3.2. Lexicaal bereik: opslaan en terugvinden van variabelen

```
1  local x
2  x = 1
3  local f
4  f = function ()  — Introductie nieuw bereik
5    x = 2 × x
6                   — Einde nieuw bereik
7  f()              — x = 2
8  f()              — x = 4
```

Stel nu dat we geen verwijzing in het bereik opslaan, maar een kopie van de omliggende bindingen. Op het moment dat we f definiëren in regel 4 is bereik s_f een paar (b_f, p_f) met $b_f \in \mathbb{B}$. 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 omliggend bereik 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 bereik s is dus een element uit de verzameling

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

Hierbij zijn $\mathbb{B}$ de bindinggroepen zoals besproken in §3.1. $\mathbb{L}_\text{s}$ zijn locaties van bereiken. Op het begrip "locatie" komen wij nog terug in §3.6. We moeten er wel rekening mee houden dat er een soort "ultiem" omliggend bereik is. Het kan dus zijn dat een bereik geen parent heeft. Voor dat geval introduceren we het unieke element $\doteq$. De beoogde interpretatie van een bereik van de vorm $(b, \doteq)$ is dan dat het geen omliggend bereik heeft.

3.3 Objecten en prototype overerving [TODO]

In §2.1.3 hebben we een beeld gekregen van prototype overerving. Net als bereik en omliggend bereik, worden objecten en hun prototypen het best gemodelleerd met een boomstructuur. Geheel in lijn met bereiken 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 $\underline{\bot}$. Een object o is dus een element uit

$$\mathbb{O} \stackrel{\text{def}}{=} \mathbb{B} \times (\mathbb{L}_\text{o} \cup \{\underline{\bot}\}).$$

Hierbij is $\mathbb{B}$ weer de verzamelingen bindinggroepen uit §3.1 en $\mathbb{L}_\text{o}$ zijn ditmaal locaties van objecten.

Hoewel het semantische model voor objecten en hun prototypen vrijwel identiek is aan het semantische model voor bereiken, hebben ze een andere betekenis. Om deze gelijkheid te benadrukken gebruiken we voor beide modellen een aantal zelfde letters, en om de ongelijkheid te benadrukken subscripten we de locatie verzamelingen ($\mathbb{L}_\text{o}$ en $\mathbb{L}_\text{s}$).

TODO

3.4 Functies [TODO]

TODO

3.5 Waarden en data

In voorgaande paragrafen spraken we over waarden $\mathbb{V}$ en locaties $\mathbb{L}$ ($\mathbb{L}_\text{s}$ voor bereiken, $\mathbb{L}_\text{o}$ voor objecten). We hebben de exacte definities hiervan in het midden gelaten. Wat wel duidelijk is dat een waarde $v \in \mathbb{V}$ is iets wat we toekennen aan een *Identifier* met behulp van een binding.

In §2.2 is uitgelegd hoe de taal met verschillende typen data omgaat. In onze taal komen namelijk drie typen *data* voor: getallen, functies en objecten. Er is echter een verschil in de manier waarop wij ze behandelen in het semantisch model. Deze verschillende aanpak ik bekend aan iedereen die een keer object-geörienteerd geprogrammeerd heeft. Zoals meeste object-geörienteerde programmeertalen maakt ook onze taal verschil tussen twee soorten data: primitieve waarden en objecten. Primitieve waarden vatten we op als de “atomen” waaruit programma’s en objecten worden opgebouwd. Denk aan natuurlijke getallen of woorden.

Code fragment 3.3. Toekenning van primitieve waarden: Getallen

```
local x
x = 5     — de primitieve waarde “5” wordt aan variabele x toegekend

local y
y = x     — nu bevat ook y de primitieve waarde “5”

x = 6     — x bevat “6”, maar y bevat nog steeds “5”
```

In onze taal zijn ook functies primitieve waarden. Zoals in 3.4 wordt beschreven, beschouwen we een functie als een “atomair” element met de volgende informatie:

- parameters
- mogelijkerwijs een return variabele
- code
- bereik van definitie

Ondanks dat dit een hoop informatie is, is het slechts een bouwsteen van complexe structuren. Het volgende voorbeeld illustreert deze gedachte:

Code fragment 3.4. Toekenning van primitieve waarden: Functies

```
local x
x = function ( ){skip}

local y
y = x

x = function (z){z = 5}   — x bevat de nieuwe functie met één argument,
                          — maar y bevat nog steeds de oude skip-functie
```

Objecten worden anders behandeld. Het idee van een object is dat dit een zekere “entiteit” voorstelt. Deze entiteit kan een nabootsing zijn van een objecten uit de reële wereld, maar kan ook iets abstracts zijn. Net als objecten uit de reële wereld, kan een object in een programma veranderen in de loop der tijd maar nog wel “hetzelfde ding zijn als tevoren”.

Het volgende code fragment illustreert dit idee:

Code fragment 3.5. Toekenning van primitieve wa

```
local x
x object
x.n = 5

local y
y = x       — y “verwijst” nu naar hetzelfde object als x

x.n = 6     — omdat x en y naar hetzelfde object verwijzen:
```

In zekere zin zijn `x` en `y` immaterieel, omdat ze op deze manier "vervangbaar" zijn. Dit in tegenstelling to `n`. Het zijn dan ook verschillende dingen: `x` en `y` zijn variabelen; `n` is een attribuut van een zeker object waar zowel `x` en `y` naar verwijzen. Het verwijzen van een variabele naar een object wordt in onze semantiek bewerkstelligd door het object ergens op te slaan, en de locatie ervan als waarde aan de variabele toe te kennen.

Om samen te vatten:

Data De wereld van data waar een programma mee om gaat bevat getallen, functies en objecten.

Waarden Een waarde is wat er in ons semantisch model aan een variabele wordt toegekend, dit met behulp van een binding. Primitieve waarden (getallen, functies) zijn zelf waarden. Om in een programma over objecten te spreken, wordt de locatie van een object ook als waarde beschouwd.

De definitie van de verzameling van alle waarden (locaties van objecten, getallen en functies) is dus als volgt:

$$\mathbb{V} \stackrel{\text{def}}{=} \mathbb{L}_\text{o} \cup \mathbb{N} \cup \mathbb{F}$$

3.6 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 $\omega \in \mathbb{L}_\text{o}$ kunnen we het bijbehorende object in een geheugen m_o opzoeken:

$$m_o(\omega).$$

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

$$\mathbb{M}_\text{o} \stackrel{\text{def}}{=} \text{“eindige”}\ \mathbb{O}^{\mathbb{L}_\text{o}}$$

3.7 Hulpfuncties

We introduceren nu een aantal hulpfuncties om de logica van bepaalde delen van het semantische model te omvatten en te beschrijven. Zoals eerder beschreven, moet de definitie van een variabele eerst gezocht worden in het huidige bereik, en daarna in omliggende bereiken. Deze procedure manifesteert zich in de hulpfunctie $\textsc{Find}_{\text{scope}} : \mathbb{M}_{\text{s}} \times \mathbb{L}_{\text{s}} \times \mathit{Identifier} \to \mathbb{L}_{\text{s}}$, die we als volgt definiëren:

$$\textsc{Find}_{\text{scope}}(m_{\text{s}}, \sigma, i) = \begin{cases} \sigma & \textbf{als } b_{m_{\text{s}}(\sigma)}(i) \neq \bot \\ \bot & \textbf{als } p_{m_{\text{s}}(\sigma)} = \doteq \\ \textsc{Find}_{\text{scope}}(m_{\text{s}}, p_{m_{\text{s}}(\sigma)}, i) & \textbf{anders} \end{cases}$$

Voor het doorzoeken van de prototype hiërarchie naar een attribuut hebben we een hulpfunctie $\textsc{Find}_{\text{proto}} : \mathbb{M}_{\text{o}} \times \mathbb{L}_{\text{o}} \times \mathit{Identifier} \to \mathbb{L}_{\text{o}}$, die identiek is aan $\textsc{Find}_{\text{scope}}$, behalve dat deze werkt met de relevante object-gerelateerde verzamelingen.

$$\textsc{Find}_{\text{proto}}(m_{\text{o}}, \omega, i) = \begin{cases} \omega & \textbf{als } b_{m_{\text{o}}(\omega)}(i) \neq \bot \\ \bot & \textbf{als } p_{m_{\text{o}}(\omega)} = \doteq \\ \textsc{Find}_{\text{proto}}(m_{\text{o}}, p_{m_{\text{o}}(\omega)}, i) & \textbf{anders} \end{cases}$$

Er is ook een hulpfunctie nodig om een pad te doorlopen. De hulpfunctie $\textsc{Trav} : \mathbb{M}_{\text{o}} \times \mathbb{L}_{\text{o}} \times \mathit{Path} \to \mathbb{L}_{\text{o}} \times \mathit{Identifier}$ levert de locatie van het laatste object op en de laatste *Identifier* van het pad (wat mogelijk naar niet noodzakelijk een attribuut is van dat laatste object), gegeven het objectgeheugen, een locatie voor het “begin” object, en een pad.

$$\textsc{Trav}(m_{\text{o}}, \omega, p) = \begin{cases} (\omega, p) & \textbf{als } p \in \mathit{Identifier} \\ \textsc{Trav}(m_{\text{o}}, \omega', p') & \textbf{als } p = i.p' \textbf{ en } \textsc{Find}_{\text{proto}}(m_{\text{o}}, \omega, i) = \omega' \in \mathbb{L}_{\text{o}} \\ \bot & \textbf{anders} \end{cases}$$

Overzicht

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

4 Natuurlijke Semantiek

4.1 Expressies

4.2 Statements

4.2.1 Basis

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

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

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 uitvoerenen daarna S_2. Tijdens dit proces ontstaan nieuwe toestanden, waar we natuurlijk rekening mee moeten houden. De geheugens worden dan ook netjes doorgesluisd.

Voor de controlestructuur `if` hebben we twee regels nodig. De eerste is voor het geval dat de *BooleanExpression* evalueert in **T**, dan moet namelijk het statement van het `then`-deel worden uitgevoerd. Wanneer de *BooleanExpression* evalueert in **F** moet het `else`-deel worden uitgevoerd. Er moet dus aan een extra voorwaarde worden voldaan om deze regels toe te mogen passen. Dit zal vaker voorkomen bij de komende deductieregels. We noteren deze extra voorwaarden onder de regel of het axioma.

$$\frac{\langle\!\langle\, S_1 \,\rangle\!\rangle_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} \longrightarrow (m'_\mathrm{s}, m'_\mathrm{o})}{\langle\!\langle\, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \,\rangle\!\rangle_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} \longrightarrow (m'_\mathrm{s}, m'_\mathrm{o})}$$
$$\textbf{desda}\quad [\![\, b \,]\!]^{\mathrm{B}}_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} = \mathbf{T}$$

en:

$$\frac{\langle\!\langle\, S_2 \,\rangle\!\rangle_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} \longrightarrow (m'_\mathrm{s}, m'_\mathrm{o})}{\langle\!\langle\, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \,\rangle\!\rangle_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} \longrightarrow (m'_\mathrm{s}, m'_\mathrm{o})}$$
$$\textbf{desda}\quad [\![\, b \,]\!]^{\mathrm{B}}_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} = \mathbf{F}$$

Eenzelfde tactiek passen we toe bij een `while`-loop.

$$\frac{\langle\!\langle\, S_1 \,\rangle\!\rangle_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} \longrightarrow (m'_\mathrm{s}, m'_\mathrm{o}) \qquad \langle\!\langle\, \texttt{while}\ b\ \texttt{do}\ S_1 \,\rangle\!\rangle_{m'_\mathrm{s},m'_\mathrm{o},\sigma,\tau} \longrightarrow (m''_\mathrm{s}, m''_\mathrm{o})}{\langle\!\langle\, \texttt{while}\ b\ \texttt{do}\ S_1 \,\rangle\!\rangle_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} \longrightarrow (m''_\mathrm{s}, m''_\mathrm{o})}$$
$$\textbf{desda}\quad [\![\, b \,]\!]^{\mathrm{B}}_{m,\sigma,\tau} = \mathbf{T}$$

en:

$$\langle\!\langle\, \texttt{while}\ b\ \texttt{do}\ S_1 \,\rangle\!\rangle_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} \longrightarrow (m_\mathrm{s}, m_\mathrm{o})$$
$$\textbf{desda}\quad [\![\, b \,]\!]^{\mathrm{B}}_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} = \mathbf{F}$$

4.2.2 Variabelen

We komen nu bij een interessanter deel van de taal, namelijk het *declareren* van variabelen en het *toekennen* van waarden. In deze sectie gaan we dus alleen de bereiken modificeren. Aan de basis hiervan ligt het declareren van een variabele met `local`. We willen in het huidige bereik de eigen bindingen zó aanpassen, dat de *Identifier* die we declareren gekoppeld wordt aan $\doteq$ (de variabele bevat immers nog geen expliciete waarde). Voordat we dit kunnen doen, moeten we bereik σ opzoeken in het geheugen voor bereiken m_s met $m_\mathrm{s}(\sigma)$. Vervolgens passen we de binding hiervan aan (zie notatie in §3.2

en §1.4). Met de verwijzing naar het omliggend bereik π doen we niks. Dit paar van uitgebreide bindingen en oud omliggend bereik zetten we terug in het geheugen m_s op plek σ.

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

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

Wanneer we daadwerkelijk een waarde aan een variabele willen toekennen, doen we dit met dezelfde aanpassingstechniek als hierboven. Er is echter één groot verschil. Voordat we een waarde aan een variabele kunnen koppelen, moeten we eerst het bereik vinden waarin deze gedeclareerd is. Dit hoeft niet het huidige bereik te zijn en dus zoeken we deze op met de hulpfunctie $\textsc{Find}_\mathrm{scope}$. Daarnaast moeten we natuurlijk de expressie aan de rechter kant van het is-teken evalueren.

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

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

4.2.3 Objecten

Bij attributen gaat toekenning net iets anders. We hebben de hulp nodig van andere functies en we moeten rekening houden met het doorlopen van een pad. Daarnaast is er nog het speciale geval dat het eerste deel van het pad **`this`** is. Natuurlijk gebeuren al deze aanpassingen in het geheugen voor objecten m_o en gebruiken we locaties van objecten $\omega \in \mathbb{L}_\mathrm{o}$ in plaats van locaties van bereiken.

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

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

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

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

Nu hebben we nog niet bekeken hoe we aangeven dat een variabele een object bevat. In §3.5 is uitgebreid besproken dat er een significant verschil is tussen primitieve waarden en objecten. Toch gaat dit op bijna dezelfde manier als het toekennen van een primitieve waarde. De locatie ω van een object o kan immers wel op dezelfde manier worden behandeld. Wat wel moet gebeuren is het aanmaken van een nieuw, leeg object in het geheugen voor objecten. Zo een leeg object heeft de vorm $(\varnothing, \perp\!\!\!\!\!\_)$. De bindingen zijn immers leeg en er is nog geen prototype gedefinieerd.

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

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

Het definiëren van een prototype gaat met het `clones` -statement. Hiervoor zoeken we simpelweg de locaties ω_i en ω_j van de twee objecten op. Vervolgens zetten we in het geheugen dat het prototype van het object in ω_i de locatie ω_j is.

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

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

4.2.4 Functies

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

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

Extra

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.

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** $\exists_\theta [\, \Box = \theta \ldots \,]$.