Een natuurlijke semantiek voor prototype overerving en lexicaal bereik

Kelley van Evert & Tim Steenvoorden

15 juni 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 op de bestaande programmeertalen JavaScript en IO. JavaScript[1] is een dynamische, prototype-gebaseerde taal die veelvuldig wordt gebruikt bij het ontwikkelen van internettoepassingen [4]. Een opvallende functie van JavaScript is het algemeen gebruik van lexicaal bereik. (Vele imperatieve talen gebruiken lexicaal bereik, maar dit vaak met beperkingen – in JavaScript is sprake van "puur" lexicaal bereik.) IO is een onderzoekstaal door Steve Dekorte [3]. 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 mensen 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:

$$\langle\!\langle\, i\ \texttt{object}\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m'_s, m'_o)$$

$$\begin{aligned}
&\textbf{desda}\ \textsc{Find}_{\text{scope}}(m_s, \sigma, i) = \sigma_{\text{def}} \\
&\qquad \omega = \textsc{Next}_{\text{object}}(m_o) \\
&\qquad m'_s = m_s\Big[\, \sigma_{\text{def}} \mapsto \Big(b_{m_s(\sigma')}[i \mapsto \omega], p_{m_s(\sigma')}\Big)\,\Big] \\
&\qquad m'_o = m_o\Big[\, \omega \mapsto \Big(\varnothing, \doteq\Big)\,\Big]
\end{aligned}$$

"Zoals jullie weten, moeten we bij statische bereiken 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

[1] Ook wel bekend als *ECMAScript*, de naam van de standaard.

twee voorgaande aan elkaar koppelt. We sluiten af met een afleiding voor een kort programma om het gebruik van de ontwikkelde regels en axioma's te demonstreren.

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 de aard van het onderwerp zullen we bijvoorbeeld vaak over *eindige*, *discrete* functies en verzamelingen spreken.

1.1 Functies

We identificeren een functie met zijn grafiek, dit wil zeggen dat een functie $f : X \to Y$ wordt gedefinieerd door de verzameling paren $(x, y) \in X \times Y$ waarvoor we beweren dat $f(x) = y$. Daarmee is een functie $f : X \to Y$ een speciaal geval van een relatie $f \subseteq X \times Y$, waarbij aan de *functionele* voorwaarde wordt voldaan dat:

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

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 gedefinieerd is op een zeker punt x, dus $\neg\exists_{y\in Y}[(x, y) \in f]$, schrijven we $f(x) = \perp$ of soms ook $f \uparrow x$. Wanneer het omgekeerde het geval is, schrijven we $f(x) \neq \perp$, of soms $f \downarrow x$, of kortweg $f(x) = y$ voor de gecombineerde uitspraak dat f wél gedefinieerd 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 gedefinieerd 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 elegante manier functies te definiëren. 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 gedefinieerd.

1.3 Eindige verzamelingen, eindige functies

De reden dat de meeste behandelde functies partieel 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 gedefinieerde variabelen in een zeker bereik. Het zou ongewoon zijn om in een programmeertaal gebruik te maken van bereiken waarin oneindig veel variabelen kunnen bestaan.

Een verzameling heet *eindig* als deze voor zekere $n \in \mathbb{N}$ gelijkmachtig is aan $\{n \in \mathbb{N} \mid n < N\}$. Een functie $f : X \to Y$ heet *eindig* wanneer de verzameling $\{x \in X \mid f \downarrow x\}$ eindig is. We schrijven in dit werkstuk Y^X voor de verzameling functies $\{f : X \to Y \mid f \text{ eindig}\}$, dit in tegenstelling tot de gebruikelijke definitie waarin Y^X alle functies $f : X \to Y$ bevat.

1.4 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 $t : \{(n, n) \in \mathbb{N}^2\} \to X$ (met X de verzameling waaruit we de elementen van de lijst nemen), waaraan nog een paar extra voorwaarden worden gesteld. We zullen ook wel $\mathbb{I}$ schrijven i.p.v. $\{(n, n) \in \mathbb{N}^2\}$, om aan te geven dat het gaat om de indices van lijsten. Merk op dat we als indices geen gebruik maken van $\mathbb{N}$. Later in het werkstuk gaan we er vanuit dat we elementen uit $\mathbb{I}$ kunnen onderscheiden van elementen uit $\mathbb{Z}$. De verzameling van alle lijsten op een zekere verzameling X, genoteerd $X_{\langle\rangle}$, is als volgt gedefinieerd:

$$X_{\langle\rangle} \stackrel{\text{def}}{=} \Big\{ t : \mathbb{I} \to X \Big| \exists_{N \in \mathbb{N}} \Big[\forall_{n<N}[t \downarrow (n, n)] \wedge \forall_{n \geq N}[t \uparrow (n, n)] \Big] \Big\}$$

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.

Een lijst t heeft *grootte*

$$|t| = N = \min\{n \in \mathbb{N} \mid t \uparrow (n, n)\}.$$

We schrijven $\langle x_0, x_1, \ldots, x_{N-1} \rangle$ voor de lijst t van grootte N met $\forall_{n<N}[t(n, n) = x_n]$. Als t een lijst is uit $X_{\langle\rangle}$, en x een element van X, dan schrijven we $t : x$, de toevoeging van x achteraan de lijst t, voor de lijst $t' = t[(N, N) \mapsto x]$, waarbij $N = |t|$.

1.5 Functie uitbreidingen

Het is vaak handig om een functie $f : X \to Y$ op een later tijdstip *uit te breiden* tot een nieuwe functie f', waarbij f' ongewijzigd blijft ten opzichte van f op alle punten $(x, y) \in f$, behalve één specifiek punt $x_1 \in X$ dat we koppelen aan $y_1 \in Y$ zodat geldt: $f'(x_1) = y_1$. De definitie van f', gebruik makend van f, noteren we als volgt:

$$f' \stackrel{\text{def}}{=} f\left[x_1 \mapsto y_1\right]$$

Waarbij f' aan de volgende eis voldoet:

$$\forall_{x \in X,\, y \in Y}\left[(x, y) \in f' \Leftrightarrow \Big((x, y) \in f \wedge x \neq x_n\Big) \vee (x, y) = (x_1, y_1)\right]$$

Meerdere uitbreidingen verkorten we op voor de hand liggende wijze:

$$f\left[x_1 \mapsto y_1,\ x_2 \mapsto y_2 \ldots x_n \mapsto y_n\right] \stackrel{\text{def}}{=} f\left[x_1 \mapsto y_1\right]\left[x_2 \mapsto y_2\right] \ldots \left[x_n \mapsto y_n\right]$$

Als $\overline{x} = \langle x_1, x_2, \ldots, x_N \rangle \in X_{\langle\rangle}$, $\overline{y} = \langle y_1, y_2, \ldots, y_N \rangle \in Y_{\langle\rangle}$, dan is bovendien

$$f\left[\overline{x} \mapsto \overline{y}\right] \stackrel{\text{def}}{=} f\left[x_1 \mapsto y_1\right]\left[x_2 \mapsto y_2\right] \ldots \left[x_N \mapsto y_N\right].$$

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 een gevoel te krijgen voor de taal. 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. De taal bevat enkel onderdelen die essentiëel zijn voor het modelleren en formaliseren van lexicaal bereik en prototype overerving. 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, minder belemmering bij het lezen van een programma.

2.1 Voorbeeldprogramma's

Elk voorbeeldprogramma en zijn toelichtingen worden als volgt gepresenteerd:

Code fragment 2.1. Het eerste voorbeeldprogramma

```
1 local f                    — f moet eerst worden gedefinieerd
2 f = function(i) returns n
3   local n
4   n = 2 × (i + 5)
5                            — x bestaat niet in dit bereik
6 local x                    — x heeft nog geen waarde, maar is wel gedefinieerd
7 x = f(42)                  — x heeft nu de waarde 94
```

De toelichtingen moeten als informeel commentaar worden beschouwd, waarmee we proberen aan 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 op andere manieren worden gebruikt. Een programma waarin variabelen worden gebruikt die nooit zijn gedefinieerd is niet valide. In code fragment 2.2 staat een voorbeeld van declaratie.

Code fragment 2.2. Declaratie van variabelen

```
1 
2 local x  — x heeft nog geen waarde, maar is wel gedefinieerd
3 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 data van drie typen:

- getallen
- functies
- objecten

Een speciale waarde ≙ (gelezen als *nul* of *niks*) wordt gebruikt om aan te geven dat een variabele nog geen data toegekend heeft gekregen. Het onderscheid tussen deze typen wordt op *dynamisch* niveau gemaakt in plaats van op syntactisch niveau. Dat houdt in dat een willekeurige variabele data van willekeurige typen kan aannemen. Ook kan het in zijn levensduur data van meerdere typen bevatten. Code fragment 2.3 geeft dit weer.

Code fragment 2.3. Data van verschillende typen

```
1 local x                — declaratie zonder type indicatie
2 x = 5                  — type "getal"
3 x = function( ){... }  — type "functie"
4 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 [4]. 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`

```
1  local x          — dit is de gezochte definitie van x
2  x = 42
3
4  local f
5  f = function( )
6     ... x ...     — waar is deze x gedefinieerd?
```

Code fragment 2.5. Zoek de definitie van variabele `x`

```
1  local x
2  x = 42
3
4  local f
5  f = function( )
6     local x       — dit is de gezochte definitie van x
7     x = 43
8     ... x ...     — waar is deze x gedefinieerd?
```

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 gedefinieerd, net als er niks gebeurt wanneer men 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 gedefinieerd. 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, ...
```

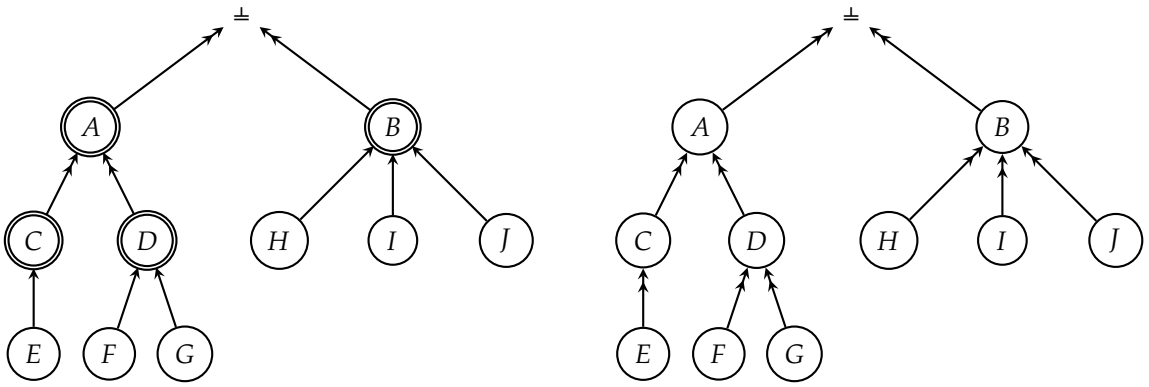
2.1.3 Prototype overerving en object oriëntatie

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 wil 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# [5, 6].

Een andere aanpak met hetzelfde doel is *prototype gebaseerde* object-oriëntatie. Daarbij wordt geen scheiding gemaakt tussen het concept *klasse*, die vorm en gedrag specificeert, en het concept *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 vorm en gedrag.

We kunnen dit principe het beste illustreren met onderstaande figuren. In figuur 2.1a zijn objecten A, B, C en D *klassen* die attributen van elkaar erven door middel van *extensie* (aangegeven met een dubbele pijl). We zeggen dat A een *superklasse* is van C en van D. Wanneer een object geen superklasse heeft, zoals A en B, geven we dit aan met het symbool $\doteq$. De overige



(a) Klassieke, klasse-gebaseerde structuur van objecten.

(b) Prototype-gebaseerde structuur van objecten.

Figuur 2.1: Twee methoden van object-geörienteerd programmeren.

objecten zijn concrete instanties van een bepaalde klasse en hebben de eigenschappen die zijn vastgelegd in die klasse. Alleen deze objecten zijn tijdens de uitvoering van het programma te manipuleren.

In figuur 2.1b zijn alle objecten instanties en dus direct te manipuleren. Daarbij erft elk object de attributen van zijn *prototype* door middel van *klonen*. In dit geval is *A* het prototype van *C* en ook van *D*. *C* is op zijn beurt weer het prototype van *E*.

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 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 het prototype van het object. Ook dan weten we de waarde en leveren deze op.
3. Wanneer ook het prototype het attribuut niet bevat, proberen we het attribuut op te zoeken in het prototype van dit object: we gaan nog een niveau hoger. We herhalen deze zoektocht voor alle volgende prototypen totdat we het attribuut hebben gevonden of een object geen prototype meer heeft.

Attributen en klonen

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`:

```
1 local Deur
2 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:

```
3 Deur.open = 1
```

Een `Deur` is standaard open. We kunnen `Deur` ook als een prototype gebruiken. In prototype-gebaseerde talen heet dit *klonen*:

```
4 local GeslotenDeur
5 GeslotenDeur object
6 GeslotenDeur clones Deur
```

Hiermee erft `GeslotenDeur` alle attributen van `Deur`. In een klassieke object-geörienteerde taal zouden we zeggen dat `Deur` nu een superklasse is van `GeslotenDeur`. Zo heeft `GeslotenDeur.open` netjes waarde 1. Maar een `GeslotenDeur` moet natuurlijk gesloten zijn. We zetten zijn attribuut `open` op 0:

```
7 GeslotenDeur.open = 0
```

Een gewone `Deur` is nog steeds open. Met andere woorden `Deur.open` heeft nog steeds 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:

```
8 local GlazenDeur
9 GlazenDeur object
10 GlazenDeur clones Deur
11 GlazenDeur.doorzichtig = 1
```

Een gewone `Deur` heeft het attribuut `doorzichtig` niet, en dus een `GeslotenDeur` ook niet. Wanneer we bijvoorbeeld `GeslotenDeur.doorzichtig` evalueren levert dit geen waarde op, dit is niet gedefinieerd. Maar we kunnen besluiten dat deuren standaard niet doorzichtig zijn:

```
12 Deur.doorzichtig = 0
```

`GeslotenDeur`en zijn nu ook niet doorzichtig. `GeslotenDeur.doorzichtig` *erft* de eigenschap van `Deur` en heeft dus waarde 0. Maar er geldt nog steeds dat `GlazenDeur.doorzichtig` waarde 1 heeft.

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. Met andere woorden, we hoeven geen klassen te definiëren en vervolgens instanties te maken van deze klassen. We creëren objecten "on-the-fly", alsmede hun attributen en relaties. Deze methode komt terug in talen als JavaScript, IO en Self [4, 3, 1].

Methoden

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:

```
13 GeslotenDeur.ontsluit = function(poging)
14   if poging = this.code then
15     this.open = 1
16   else
17     this.open = 0
```

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

```
18 local Kluis
19 Kluis object
20 Kluis clones GeslotenDeur
21 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:

```
22 Kluis.ontsluit(1234)
```

Wanneer we nu `Kluis.open` evalueren komt hier 0 uit, we hebben de verkeerde code ingevoerd! We proberen het nog een keer:

```
23 Kluis.ontsluit(4321)
```

Deze code is wel correct en `Kluis.open` heeft nu 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 gedefinieerd:

$$\mathit{Number} ::= (-)^? \ (0 \mid 1)^+$$

Eigenlijk gebruiken we geen strikte BNF, in deze specifieke gevallen, maar een hele simpele variant, zoals E-BNF [2], die ook reguliere expressies toelaat. Bovenstaand voorbeeld maakt dit duidelijk. Voorbeelden van elementen uit *Number* zijn "0", "1", "0111001" en "0001". Voorbeelden van elementen die niet in *Number* zitten zijn "", "-6", "4.2". Hoewel we formeel in het binaire stelsel werken, zullen we in voorbeeldprogramma's wel gebruiken maken van het decimale stelsel.

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

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

Hierbij moet men zich voorstellen dat alle letters uit het alfabet in de grammaticaregel staan op de voor de hand liggende manier. Merk op dat we geen gebruik maken van getallen of streepjes binnen bovenstaande namen.

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:

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

We introduceren geen speciale syntaxis om naar het **`this`** object te refereren, maar zullen een pad dat begint met de *Identifier***`this`** beschouwen als een pad binnen het **`this`** object. Merk op dat als een `this` op een andere plek in een pad wordt gebruikt, er wel gewoon gesproken wordt over een zeker `this`-attribuut van een zeker object: alleen vooraan een pad is een speciale betekenis van toepassing.

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:

$$
\begin{aligned}
\mathit{Expression} ::= {} & \mathit{Number} \mid \mathit{Expression}\ (+ \mid - \mid \times)\ \mathit{Expression} \mid \mathit{Path} \\
\mid {} & \texttt{function}\,(\,\mathit{MaybeIdentifiers}\,)\ (\texttt{returns}\ \mathit{Identifier})^{?}\ \{\ \mathit{Statement}\ \} \\
\mathit{BooleanExpression} ::= {} & \texttt{true} \mid \texttt{false} \\
\mid {} & \mathit{BooleanExpression}\ (\texttt{and} \mid \texttt{or})\ \mathit{BooleanExpression} \\
\mid {} & \texttt{not}\ \ \mathit{BooleanExpression} \\
\mid {} & \mathit{Expression}\ (= \mid \neq \mid < \mid \leq \mid > \mid \geq)\ \mathit{Expression}
\end{aligned}
$$

Ook hier is het handig om twee toevoegingen te maken op *Expression*:

$$
\begin{aligned}
\mathit{Expressions} &::= \mathit{Expression} \mid \mathit{Expressions}, \mathit{Expression} \\
\mathit{MaybeExpressions} &::= \varepsilon \mid \mathit{Expressions}
\end{aligned}
$$

De productieregel voor *statements* is de kern van de grammatica. 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 de syntactische verzameling *Statement*. (Vanwege de focus van dit werkstuk definiëren we niet precies wanneer een programma valide is en wanneer niet.)

$$
\begin{aligned}
\mathit{Statement} ::= \ & \texttt{skip} \\
| \ & \mathit{Statement}\texttt{;}\ \mathit{Statement} \\
| \ & \texttt{if}\ \mathit{BooleanExpression}\ \texttt{then}\ \mathit{Statement}\ \texttt{else}\ \mathit{Statement} \\
| \ & \texttt{while}\ \mathit{BooleanExpression}\ \texttt{do}\ \mathit{Statement} \\
| \ & \texttt{local}\ \mathit{Identifier} \\
| \ & \mathit{Identifier}\ \texttt{object} \\
| \ & \mathit{Identifier}\ \texttt{clones}\ \mathit{Identifier} \\
| \ & \mathit{Path} = \mathit{Expression} \\
| \ & (\mathit{Path} =)^{?}\ \mathit{Identifier}\texttt{(}\ \mathit{MaybeExpressions}\texttt{)}
\end{aligned}
$$

Merk op dat in delen van deze productieregel soms een *Identifier* staa 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 om dit niet te doen. Allereerst worden de axioma's en deductieregels ingewikkelder en daarmee minder elegant en 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 gedefinieerde variabelen binnen een bereik vast te leggen of attributen van een bepaald object. Een *groep bindingen* is een eindige functie $b : \mathit{Identifier} \to (\mathbb{V} \cup \{\doteq\})$. We gebruiken de "speciale waarde" $\doteq$ om aan te geven dat een variabele wel gedefinieerd is, maar nog geen concrete waarde toegekend heeft gekregen. De verzameling van alle groepen van bindingen definiëren we als

$$\mathbb{B} \stackrel{\text{def}}{=} (\mathbb{V} \cup \{\doteq\})^{\mathit{Identifier}}$$

We komen in §3.5 terug op wat de waarden $\mathbb{V}$ precies zijn. Voor nu is het voldoende om te weten dat in ieder geval de gehele getallen $\mathbb{Z}$ 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. Bij het uitbreiden van bindinggroepen gebruiken we de notatie zoals gedefinieerd in §1.5.

3.2 Bereik en omliggend bereiken

In sectie 2.1 is informeel gebleken dat bereiken conceptueel goed te zien zijn als een boomstructuur. Stel dat we een variabele x in een zeker bereik s willen evalueren tot een waarde. Dan zoeken we x eerst op in de bindingen groep b_s, behorende bij bereik s met $b_s(x)$. Vervolgens moet er gevalsonderscheiding worden gedaan voor de volgende situaties:

1. x is gedefinieerd in b_s, dus gebruiken we de gevonden waarde.
2. x is niet gedefinieerd in b_s, dus moeten we x opzoeken in het 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 groep en π een *verwijzing* naar het 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 3.1 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. Het omliggend bereik π_f van functie f verwijst naar bereik s. Hierin is x wel **`local`** gedefinieerd, zodat de variabele uiteindelijk gevonden wordt.

Code fragment 3.1. Lexicaal bereik: opslaan en terugvinden van variabelen

```
1  local x          — Omliggend bereik s = (b_s, π_s).
2  x = 1
3  local f
4  f = function()   — Introductie nieuw bereik s_f = (b_f, π_f)
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, π_f) met $b_f, \pi_f \in \mathbb{B}$. Net als hierboven zijn de eigen bindingen b_f leeg. De binding π_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

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

Hierbij zijn $\mathbb{B}$ de bindinggroepen zoals besproken in §3.1. $\mathbb{I}$ is de verzameling lijst-indices, zie §1.4. Zo'n index is de wiskundige verwezenlijking van een *referentie*, waarvan het gebruik zoals eerder beschreven essentiëel is. In §3.6 wordt behandeld hoe bereiken en objecten met behulp van lijsten het geheugen vormen.

We moeten er rekening mee houden dat er een soort "ultiem" omliggend bereik is. Een stuk code is namelijk eindig, en wanneer het uitgevoerd wordt is er sprake van een buitenste bereik. Het kan dus zijn dat een bereik geen parent heeft. Voor dat geval gebruiken we het unieke element $\doteq$. De beoogde interpretatie van een bereik van de vorm $(b, \doteq)$ is dan dat het geen omliggend bereik heeft, en dus het buitenste bereik van de code in executie.

3.3 Objecten en prototype overerving

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 géén prototype hebben. Dit geven we weer aan met $\doteq$. Een object o is dus een element uit

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

3.4 Functies

Van een functie moeten een aantal dingen bekend zijn:

Code De code (ook wel *body*) van de functie is natuurlijk simpelweg een *Statement*.

Parameters De parameters van een functie worden gerepresenteerd door een lijst elementen uit *Identifier*. In het programma zelf worden de argumenten nog gerepresenteerd door een element uit *MaybeIdentifiers*. We definiëren daarom de triviale functie $\textsc{Ids} : \mathit{MaybeIdentifiers} \to \mathit{Identifier}_{\langle\rangle}$ op de volgende manier:

$$\begin{aligned} \textsc{Ids}(\varepsilon) &= \langle\rangle \\ \textsc{Ids}(i) &= \langle i \rangle \\ \textsc{Ids}(i^+, i) &= \textsc{Ids}(i^+) : i \end{aligned}$$

Hierbij is i^+ een meta-variabele voor een element uit de syntactische verzameling *Identifiers*.

Return variabele Mogelijkerwijs geeft de functie ook een resultaat terug. In onze taal gebeurt dit door bij definitie van de functie aan te geven van welke variabele, die gevonden moet kunnen worden in het gemaakt bereik, de waarde terug gegeven moet worden. Dit wordt gerepresenteerd door een element uit $(\mathit{Identifier} \cup \{\doteq\})$, waarbij $\doteq$ de betekenis draagt dat

er geen sprake is van een resultaat. De functie heeft dan nog wel betekenis, omdat het zij-effecten kan voortbrengen.

Bereik van definitie Het is belangrijk om van een functie zijn bereik van definitie te weten. Dit is het kern-idee van lexicaal bereik. Het kan zijn dat een zekere functie f op een zekere lexicale plek in het programma is gedefinieerd, en vervolgens door toedoen van het programma in een andere context wordt uitgevoerd. Belangrijk is dat het lexicale bereik van de originele definitie nog bekend is, om vrije variabelen in de code van de functie op te kunnen zoeken in juiste omliggende bereiken. Dit bereik wordt gerepresenteerd door de locatie van het bereik, een element uit $\mathbb{I}$.

Een functie is daarom simpelweg een viertupel met bovenstaande informatie. De verzameling functies is gedefinieerd als:

$$\mathbb{F} \stackrel{\text{def}}{=} \mathit{Statement} \times \mathit{Identifier}_{\langle\rangle} \times (\mathit{Identifier} \cup \{\doteq\}) \times \mathbb{I}$$

3.5 Waarden en data

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

In §2.1.1 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. Dit verschil in aanpak komt terug in de meeste object-geörienteerde talen [5, 6]. Zo maken wij ook onderscheid 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 getallen of woorden.

Code fragment 3.2. Toekenning van primitieve waarden: Getallen

```
1  local x
2  x = 5    — de primitieve waarde "5" wordt aan variabele x toegekend
3
4  local y
5  y = x    — nu bevat ook y de primitieve waarde "5"
6
7  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 waarvan de parameters, mogelijkerwijs een teruggeef

variabele, code en bereik van definitie bekend zijn. Ondanks dat dit een hoop informatie is, is het slechts een bouwsteen van complexere structuren. Het volgende voorbeeld illustreert deze gedachte:

Code fragment 3.3. Toekenning van primitieve waarden: Functies

```
1  local x
2  x = function( ){skip}
3
4  local y
5  y = x
6
7  x = function(z){z = 5}  — x bevat de nieuwe functie met één argument,
8                          — 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 3.4. Toekenning van primitieve waarden: Objecten

```
1  local x
2  x object
3  x.n = 5
4
5  local y
6  y = x      — y "verwijst" nu naar hetzelfde object als x
7
8  x.n = 6    — omdat x en y naar hetzelfde object verwijzen: y.n = 6
```

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. Wanneer een variabele nog geen waarde toegekend heeft gekregen, bevat het nog geen waarde: $\doteq$.

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

$$\mathbb{V} \stackrel{\text{def}}{=} \mathbb{I} \cup \mathbb{Z} \cup \mathbb{F}$$

3.6 Geheugen

Uit de redenaties over referenties volgt dat we op een bepaalde manier objecten moeten kunnen opslaan aan de hand van deze referenties. De waarden van referenties zelf zijn niet belang, enkel hun uniciteit. We definiëren daarom het *geheugen voor objecten* m_o simpelweg als een lijst objecten, waarbij de indices als referenties dienen. Ook moet een geheugen van bereiken m_s worden bijgehouden. De verzamelingen van deze twee typen geheugens zijn dus:

$$\mathbb{M}_\text{o} \stackrel{\text{def}}{=} \mathbb{O}_{\langle\rangle} \qquad\qquad \mathbb{M}_\text{s} \stackrel{\text{def}}{=} \mathbb{S}_{\langle\rangle}$$

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 $\textsc{Find}_\text{scope} : \mathbb{M}_\text{s} \times \mathbb{I} \times \mathit{Identifier} \to \mathbb{I}$, die we als volgt definiëren:

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

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

relevante object-gerelateerde verzamelingen.

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

Er is ook een hulpfunctie nodig om een pad te doorlopen. De hulpfunctie $\textsc{Trav} : \mathbb{M}_{\text{o}} \times \mathbb{I} \times \mathit{Path} \to \mathbb{I} \times \mathit{Identifier}$ 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.

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

4 Natuurlijke Semantiek

4.1 Expressies

Expressies zijn syntactische elementen die zonder de *toestand* van het programma te wijzigen *geëvalueerd* kunnen worden tot een waarde $v \in \mathbb{V}$. Voor het evalueren definiëren we een aantal semantische functies, en één semantisch predikaat. Met een notatie die gebruik maakt van dubbele rechte haken geven we aan dat hier de stap wordt gemaakt van de *syntaxis* van de taal naar het semantische model. Deze semantische functies (predikaat) nemen als input, naast de expressie die geëvalueerd dient te worden, ook de toestand van het programma: de geheugens m_s en m_o voor bereiken en objecten. Daarnaast leggen het huidige bereik σ en het huidige object τ vast in welke *omgeving* de evaluatie plaats vindt. Deze extra informatie noteren we als subscript bij de haken. Zo zijn evaluaties van "gewone" expressies (*Expression*) zijn van de vorm

$$[\![E]\!]_{m_\mathrm{s}, m_\mathrm{o}, \sigma, \tau} = v \in \mathbb{V}.$$

De signatuur van deze functie $[\![\ ]\!]$ is dan

$$[\![\ ]\!] : \mathit{Expression} \times \mathbb{M}_\mathrm{s} \times \mathbb{M}_\mathrm{o} \times \mathbb{L}_\mathrm{s} \times \mathbb{L}_\mathrm{o} \to \mathbb{V}$$

Ook definiëren we een *uitgebreide* semantische functie voor syntactische elementen uit *MaybeExpressions*. Deze functie is nodig wanneer een functie wordt aangeroepen met een willekeurig aantal argumenten. Merk op dat in dit geval het resultaat niet een enkele waarde is, maar een lijst van waarden.

$$[\![\ ]\!]^\mathrm{M} : \mathit{MaybeExpressions} \times \mathbb{M}_\mathrm{s} \times \mathbb{M}_\mathrm{o} \times \mathbb{L}_\mathrm{s} \times \mathbb{L}_\mathrm{o} \to \mathbb{V}_{\langle\rangle}$$

4.1.1 Een predikaat voor boolse expressies

Boolse expressies, die gebruikt worden in de verscheidene loops van de taal, evalueren we niet tot een zekere verzameling. In tegenstelling tot gewone expressies, definiëren we voor boolse

expressies een predikaat $[\![\ ]\!]^{\mathrm{B}}$.

$$[\![\ ]\!]^{\mathrm{B}} \subseteq \mathit{BooleanExpression} \times \mathbb{M}_{\mathrm{s}} \times \mathbb{M}_{\mathrm{o}} \times \mathbb{L}_{\mathrm{s}} \times \mathbb{L}_{\mathrm{o}}$$

De reden voor deze andere aanpak is dat de definitie dan makkelijker leest. We definiëren dat $[\![b]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau}$, dan en slechts dan als het met de volgende axioma's en deductieregels bewezen kan worden:

$$[\![\texttt{true}]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau} \qquad \frac{\neg [\![b]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau}}{[\![\texttt{not}\ b]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau}}$$

$$\frac{[\![b]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau} \quad [\![c]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau}}{[\![b\ \texttt{and}\ c]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau}} \qquad \frac{[\![b]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau}}{[\![b\ \texttt{or}\ c]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau}} \qquad \frac{[\![c]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau}}{[\![b\ \texttt{or}\ c]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau}}$$

$$[\![e_1 \sim e_2]\!]^{\mathrm{B}}_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau} \ \textbf{desda}\ n_1 \sim n_2$$
$$\textbf{waarbij}\ n_1 = [\![e_1]\!]_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau} \in \mathbb{Z}$$
$$n_2 = [\![e_2]\!]_{m_{\mathrm{s}},m_{\mathrm{o}},\sigma,\tau} \in \mathbb{Z}$$
$$(\sim)\ \textbf{één van:}\ =, \neq, <, \leq, >, \geq$$

4.1.2 Evaluatie van gehele getallen

Het evalueren van getallen (*Number*) beschouwen we voor het gemak apart, en nemen we later op in de volledige semantische functie voor expressies (*Expression*).

$$[\![\ ]\!]^{\mathrm{Z}} : \mathit{Number} \to \mathbb{Z}$$

Deze functie is als volgt op voor de hand liggend manier gedefinieerd:

$$[\![0]\!]^{\mathrm{Z}} = 0 \qquad [\![n0]\!]^{\mathrm{Z}} = 2 \cdot [\![n]\!]^{\mathrm{Z}}$$
$$[\![1]\!]^{\mathrm{Z}} = 1 \qquad [\![n1]\!]^{\mathrm{Z}} = 2 \cdot [\![n]\!]^{\mathrm{Z}} + 1$$
$$[\![-n]\!]^{\mathrm{Z}} = -1 \cdot [\![n]\!]^{\mathrm{Z}}$$

4.1.3 Evaluatie van expressies

De eerste type expressie dat we tegenkomen in de definitie van §2.2 zijn de getallen. Deze hebben we hierboven al gedefinieerd met de gespecialiseerde functie $[\![\ ]\!]^{\mathbb{Z}}$. Hier kunnen we nu mooi gebruik van maken.

$$[\![n]\!]_{m_s,m_o,\sigma,\tau} = [\![n]\!]^{\mathbb{Z}} \qquad \text{[num]}$$

Daarnaast hebben we natuurlijk de aritmetische expressies van de form $e_1 \circ e_2$. Hierbij is $\circ$ een van de operatoren $+$, $-$ of $\times$. Aritmetische expressies worden geëvalueerd door eerst beide deelexpressies te evalueren tot getallen en hier vervolgens de betreffende operator op lost te laten.

$$[\![e_1 \circ e_2]\!]_{m_s,m_o,\sigma,\tau} = n_1 \circ n_2 \qquad \text{[op]}$$
$$\textbf{waarbij } n_1 = [\![e_1]\!]_{m_s,m_o,\sigma,\tau} \in \mathbb{Z}$$
$$n_2 = [\![e_2]\!]_{m_s,m_o,\sigma,\tau} \in \mathbb{Z}$$
$$(\circ) \textbf{ één van: } +, -, \times$$

Bij de evaluatie van een *Path*-expressie moeten we rekening houden met twee dingen. Ten eerste is een pad gedefinieerd als een losse *Identifier* of een aaneenschakeling van de vorm *Identifier.Path*. Ten tweede kan de eerste *Identifier* die we tegenkomen een verwijzing zijn naar het huidige object: `this`. Het combineren van deze twee aandachtspunten levert ons vier verschillende gevallen op die we onderscheiden met *pattern matching*.

Het eenvoudigste geval is wanneer we te maken hebben met één *Identifier* die gelijk is aan `this`. We retourneren simpelweg de locatie van het huidige object τ.

$$[\![\texttt{this}]\!]_{m_s,m_o,\sigma,\tau} = \tau \qquad \text{[this]}$$

Wanneer we te maken hebben met een willekeurige andere *Identifier* i, schakelen we de hulp in van de functie $\textsc{Find}_{\text{scope}}$. Zoals beschreven in §3.7 zoekt deze het bereik σ' op waarin i gedefinieerd is (mogelijk gelijk aan σ zelf). Vervolgens zoeken we de waarde van i op in de bindingen van dit bereik.

$$[\![i]\!]_{m_s,m_o,\sigma,\tau} = v \qquad \text{[identifier]}$$
$$\textbf{desda } \textsc{Find}_{\text{scope}}(m_s, \sigma, i) = \sigma'$$
$$b_{m_s(\sigma')}(i) = v$$

De volgende stap is het geval waarin we te maken hebben met een *Path* p in plaats van een losse *Identifier*. Hier schakelen we als eerste $\textsc{Trav}$ in, die het pad doorloopt en ons het laatst

ontdekte object ω' en de laatste *Identifier* j teruggeeft. Hierna kunnen we op zoek naar het prototype ω'' waarin j is gedefinieerd met hulp van $\textsc{Find}_{\text{proto}}$. Tot slot zoeken we de waarde van j op in de bindingen van dit prototype. Wanneer het eerste onderdeel van het *Path* begint met `this` doorlopen we het pad beginnende bij het huidige object τ.

$$[\![\ \texttt{this}\ .p\ ]\!]_{m_s,m_o,\sigma,\tau} = v \qquad \text{[this.path]}$$
$$\textbf{desda}\ \textsc{Trav}(m_o,\tau,p) = (\omega' \in \mathbb{I}, j)$$
$$\textsc{Find}_{\text{proto}}(\omega', j) = \omega''$$
$$b_{m_o(\omega'')}(j) = v$$

Als dit niet het geval is, moeten we eerst op zoek naar het object in i. Hier kunnen we handig gebruik maken van de regel [identifier] hierboven.

$$[\![\ i.p\ ]\!]_{m_s,m_o,\sigma,\tau} = v \qquad \text{[path]}$$
$$\textbf{desda}\ [\![\ i\ ]\!]_{m_s,m_o,\sigma,\tau} = \omega \in \mathbb{I}$$
$$\textsc{Trav}(m_o,\omega,p) = (\omega' \in \mathbb{I}, j)$$
$$\textsc{Find}_{\text{proto}}(\omega', j) = \omega''$$
$$b_{m_o(\omega'')}(j) = v$$

Merk op dat wanneer $[\![\ i\ ]\!]_{m_s,m_o,\sigma,\tau}$ geen $\omega \in \mathbb{I}$ oplevert bovenstaande niet gedefinieerd is en $[\![\ i.p\ ]\!]$ dus niet geëvalueerd kan worden.

De laatste familie van expressies zijn de functies. Hier maken we onderscheid tussen functies *met* teruggeefwaarde en *zonder* teruggeefwaarde. In §3.4 is besproken hoe wij functies opbouwen in ons model. Onderstaande expressies creëren zo'n functie.

$$[\![\ \texttt{function}(i^*)\ \{S\}\ ]\!]_{m_s,m_o,\sigma,\tau} = (S, \textsc{Ids}(i^*), \doteq, \sigma) \qquad \text{[function]}$$

$$[\![\ \texttt{function}(i^*)\ \texttt{returns}\ j\ \{S\}\ ]\!]_{m_s,m_o,\sigma,\tau} = (S, \textsc{Ids}(i^*), j, \sigma) \qquad \text{[function returns]}$$

We zien dat we alle benodigde gegevens uit de expressie overnemen en opslaan in een $f \in \mathbb{F}$. Het enige verschil tussen de regels [function] en [function returns] is het overnemen van de teruggeef-*Identifier* j in het viertupel.

De volgende voor de hand liggende definitie evalueert een element uit de syntactische verza-

meling *MaybeExpressions* to een lijst van waarden.

$$\begin{aligned}
[\![\ \varepsilon\ ]\!]^{\mathrm{M}}_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} &= \langle\rangle \\
[\![\ E\ ]\!]^{\mathrm{M}}_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} &= \langle [\![\ E\ ]\!]_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} \rangle \\
[\![\ E^+, E\ ]\!]^{\mathrm{M}}_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} &= [\![\ E^+\ ]\!]^{\mathrm{M}}_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau} : [\![\ E\ ]\!]_{m_\mathrm{s},m_\mathrm{o},\sigma,\tau}
\end{aligned}$$

Hier is E^+ een meta-variabele voor elementen uit *Expressions*.

4.2 Statements

4.2.1 Basis

Het evalueren van een statement noteren we met dubbele vishaken. Net als bij expressies hebben we ook hier extra informatie nodig in de vorm van m_s, m_o, σ en τ. Ook hier noteren we deze als subscript bij de haken. Onze uitspraken zijn dan 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

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

Deze transitie werkt op een statement $S \in \mathit{Statement}$ in een toestand $(m_\mathrm{s}, m_\mathrm{o}) \in (\mathbb{M}_\mathrm{s}, \mathbb{M}_\mathrm{o})$, waarbij ook bekend is wat het huidige bereik $\sigma \in \mathbb{I}$, en het huidige object $\tau \in (\mathbb{I} \cup \{\doteq\})$ is. Het resultaat is een nieuwe toestand in de vorm van de twee geheugens $(m'_\mathrm{s}, m'_\mathrm{o}) \in (\mathbb{M}_\mathrm{s}, \mathbb{M}_\mathrm{o})$.

Het huidige bereik verandert niet na executie van een programma, omdat bereik te maken heeft met de ordening van de code van het programma, welke niet verandert tijdens executie. Ook het huidige object verandert niet na executie, omdat de executie van een programma binnen een bepaald object uitvoering van een methode binnen dit object representeert. Het huidige bereik en huidig object geven dus de context van executie weer, maar niet de toestand van een programma.

Het kan zijn dat er geen huidig object is, dan is $\tau = \doteq$. Dit is bijvoorbeeld van toepassing op de executie van de eerste regel van een programma, dat niet deel is van een andere programma

(bijvoorbeeld de body van een functie).

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

$$\langle\!\langle\ \texttt{skip}\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m_s, m_o) \qquad \text{[skip]}$$

`skip` verandert niets aan de toestand zodat $(m'_s, m'_o) = (m_s, m_o)$. Voor het samenstellen van statements hebben we de volgende regel nodig.

$$\frac{\langle\!\langle\ S_1\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m'_s, m'_o) \quad \langle\!\langle\ S_2\ \rangle\!\rangle_{m'_s,m'_o,\sigma,\tau} \longrightarrow (m''_s, m''_o)}{\langle\!\langle\ S_1; S_2\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m''_s, m''_o)} \qquad \text{[comp]}$$

Deze regel geeft aan dat het effect van twee sequentiele statements het gecombineerde effect is van de afzonderlijke, sequentiele, executie van beide statements.

Voor de controlestructuur **`if`** hebben we twee regels nodig. De eerste is van toepassing als de boolse expressie en ware uitspraak over de huidige toestand van het programma representeerd, waarna het **`then`**-deel moet worden uitgevoerd. De tweede als dit niet het geval is, waarna het **`else`**-deel moet 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.

$$\frac{\langle\!\langle\ S_{\text{then}}\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m'_s, m'_o)}{\langle\!\langle\ \texttt{if}\ b\ \texttt{then}\ S_{\text{then}}\ \texttt{else}\ S_{\text{else}}\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m'_s, m'_o)} \quad \textbf{desda}\ [\![\ b\ ]\!]^{\mathrm{B}}_{m_s,m_o,\sigma,\tau} \qquad \text{[if \#1]}$$

$$\frac{\langle\!\langle\ S_{\text{else}}\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m'_s, m'_o)}{\langle\!\langle\ \texttt{if}\ b\ \texttt{then}\ S_{\text{then}}\ \texttt{else}\ S_{\text{else}}\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m'_s, m'_o)} \quad \textbf{desda}\ \neg[\![\ b\ ]\!]^{\mathrm{B}}_{m_s,m_o,\sigma,\tau} \qquad \text{[if \#2]}$$

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

$$\frac{\langle\!\langle\ S_{\text{while}}\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m'_s, m'_o) \quad \langle\!\langle\ \texttt{while}\ b\ \texttt{do}\ S_{\text{while}}\ \rangle\!\rangle_{m'_s,m'_o,\sigma,\tau} \longrightarrow (m''_s, m''_o)}{\langle\!\langle\ \texttt{while}\ b\ \texttt{do}\ S_{\text{while}}\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m''_s, m''_o)} \quad \textbf{desda}\ [\![\ b\ ]\!]^{\mathrm{B}}_{m,\sigma,\tau} \qquad \text{[while \#1]}$$

$$\langle\!\langle\ \texttt{while}\ b\ \texttt{do}\ S\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m_s, m_o) \qquad \text{[while \#2]}$$
$$\textbf{desda}\ \neg[\![\ b\ ]\!]^{\mathrm{B}}_{m_s,m_o,\sigma,\tau}$$

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 $m_s(\sigma)$. Vervolgens passen we de binding hiervan aan (zie notatie in §3.2 en §1.5). 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.

$$\langle\!\langle\ \texttt{local}\ i\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m'_s, m_o) \qquad \text{[local]}$$
$$\textbf{desda}\ m'_s = m_s\big[\ \sigma \mapsto \big(b_{m_s(\sigma)}[i \mapsto \doteq], \pi_{m_s(\sigma)}\big)\ \big]$$

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 $\textsc{Find}_{\text{scope}}$. Daarnaast moeten we natuurlijk de expressie aan de rechter kant van het is-teken evalueren.

$$\langle\!\langle\ i = e\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m'_s, m_o) \qquad \text{[assign var]}$$
$$\begin{aligned}\textbf{desda}\ &[\![\ e\ ]\!]_{m_s,m_o,\sigma,\tau} = v\\ &\sigma' = \textsc{Find}_{\text{scope}}(m_s, \sigma, i)\\ &m'_s = m_s\big[\ \sigma' \mapsto \big(b_{m_s(\sigma')}[i \mapsto v], \pi_{m_s(\sigma')}\big)\ \big]\end{aligned}$$

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 in plaats van het geheugen voor bereiken m_s.

$$\langle\!\langle\ \texttt{this}\ .p = e \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m_s, m_o') \qquad \text{[assign this attr]}$$
$$\textbf{desda}\ [\![e]\!]_{m_s,m_o,\sigma,\tau} = v$$
$$\textsc{Trav}(m_o, \tau, p) = (\omega, i)$$
$$m_o' = m_o\big[\, \omega \mapsto \big(b_{m_o(\omega)}[i \mapsto v], \pi_{m_o(\omega)}\big)\,\big]$$

$$\langle\!\langle\ i.p = e \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m_s', m_o) \qquad \text{[assign attr]}$$
$$\textbf{desda}\ [\![e]\!]_{m_s,m_o,\sigma,\tau} = v$$
$$[\![i]\!]_{m_s,\sigma,\tau} = \omega$$
$$\textsc{Trav}(m_o, \omega, p) = (\omega', j)$$
$$m_o' = m_o\big[\, \omega' \mapsto \big(b_{m_s(\omega')}[j \mapsto v], \pi_{m_s(\omega')}\big)\,\big]$$

Nu hebben we nog niet bekeken hoe we aangeven dat een variabele een object bevat data. In §3.5 is uitgebreid besproken dat er een significant verschil is tussen primitieve waarden en data. 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 moet gebeuren is het aanmaken van een nieuw, leeg object in het geheugen voor objecten. Zo een leeg object heeft de vorm $(\varnothing, \triangleq)$. De bindingen zijn immers leeg en er is nog geen prototype gedefinieerd. Om een nieuwe plek in het geheugen te vinden, kijken we naar de lengte van het huidge geheugen. Tot slot voegen we het legen object achteraan het geheugen toe.

$$\langle\!\langle\ i\ \texttt{object}\ \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m_s', m_o') \qquad \text{[object]}$$
$$\textbf{desda}\ \textsc{Find}_{\text{scope}}(m_s, \sigma, i) = \sigma'$$
$$\omega = |m_o|$$
$$m_s' = m_s\big[\, \sigma' \mapsto \big(b_{m_s(\sigma')}[i \mapsto \omega], \pi_{m_s(\sigma')}\big)\,\big]$$
$$m_o' = m_o : (\varnothing, \triangleq)$$

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.

$$\langle\!\langle\ i\ \texttt{clones}\ j \rangle\!\rangle_{m_s,m_o,\sigma,\tau} \longrightarrow (m_s, m_o') \qquad \text{[clones]}$$
$$\textbf{desda}\ [\![i]\!]_{m_s,m_o,\sigma,\tau} = \omega_i \in \mathbb{I}$$
$$[\![j]\!]_{m_s,m_o,\sigma,\tau} = \omega_j \in \mathbb{I}$$
$$m_o' = m_o\big[\, \omega_i \mapsto \big(b_{m_o(\omega_i)}, \omega_j\big)\,\big]$$

4.2.4 Functie applicatie

Functie applicatie is de meest interessante deductieregel, en de kern van de semantiek als het om lexicaal bereik gaat. Een viertal varianten kunnen van toepassing zijn, om de volgende twee redenen:

- Naar de functie die aangeroepen wordt kan worden verwezen met een variabele of een attribuut. Als het eerste het geval is, dan wordt de body van de functie uitgevoerd zonder huidig object. Als het tweede het geval is, wordt het huidig object gekozen als het object waarvan de attribuut is. Deze ogenschijnlijk onnodig gecompliceerde werking is geïnspireerd door JavaScript, en biedt eigenlijk grote flexibiliteit.
- Het kan zijn dat de functie een waarde oplevert, maar dit hoeft niet het geval te zijn.

$$\frac{\langle\!\langle\, S_{\text{body}} \,\rangle\!\rangle_{m'_s, m_o, \sigma_{\text{exec}}, \tau_{\text{exec}}} \longrightarrow (m''_s, m'_o)}{\langle\!\langle\, (i_{\text{receive}} =)^{?}\ s_{\text{refer}}(e^*) \,\rangle\!\rangle_{m_s, m_o, \sigma, \tau} \longrightarrow (m'''_s, m'_o)} \quad \text{[apply]}$$

$$\textbf{desda}\ [\![\, s \,]\!]_{m_s, m_o, \sigma, \tau} = f = \langle S_{\text{body}}, I_{\text{params}}, r, \sigma_{\text{def}} \rangle \in \mathbb{F}$$

$$\tau_{\text{exec}} = \begin{cases} \omega' \textbf{ als } \begin{bmatrix} s_{\text{refer}} = i.s \\ \sigma_{\text{def}} = \textsc{Find}_{\text{scope}}(m_s, \sigma, i) \\ b_{m_s(\sigma_{\text{def}})}(i) = \omega \in \mathbb{I} \\ \textsc{Trav}(m_o, \omega, s) = (\omega', j) \end{bmatrix} \\ \triangleq \textbf{ als } s_{\text{refer}} = i \end{cases}$$

$$\sigma_{\text{exec}} = |m_s|$$

$$m'_s = m_s : \langle \varnothing \big[\textsc{Ids}(I_{\text{params}}) \mapsto [\![\, e^* \,]\!]^{\text{M}}_{m_s, m_o, \sigma, \tau} \big], \sigma_{\text{def}} \rangle$$

$$m'''_s = \begin{cases} m''''_s \textbf{ als } \begin{bmatrix} i_{\text{receive}} \text{ in de syntax staat} \\ r = i_{\text{return}} \in \textit{Identifier} \\ \sigma_{\text{rd}} = \textsc{Find}_{\text{scope}}(m''_s, \sigma, i_{\text{receive}}) \\ v = [\![\, i_{\text{return}} \,]\!]_{m''_s, m'_o, \sigma_{\text{exec}}, \tau_{\text{exec}}} \\ m''''_s = m''_s \big[\, \sigma_{\text{rd}} \mapsto \big(b_{m''_s(\sigma_{\text{rd}})}[i_{\text{receive}} \mapsto v], \pi_{m''_s(\sigma_{\text{rd}})} \big) \,\big] \end{bmatrix} \\ m''_s \textbf{ als } r = \triangleq \end{cases}$$

Bibliografie

[1] Russell Allen. *Self Handbook Documentation*, August 2011. URL `http://docs.selflanguage.org/4.4/pdf/SelfHandbook-Self4.4-R2.pdf`. Release 2 for Self 4.4.

[2] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0.* W3C, 5 edition, November 2008. URL `http://www.w3.org/TR/xml/#sec-notation`. Alleen de sectie over E-BNF-notatie.

[3] Steve Dekorte. *Io Programming Guide*, 2010. URL `http://www.iolanguage.com/scm/io/docs/IoGuide.html`.

[4] *ECMAScript Language Specification.* ECMA International, 5.1 edition, June 2011. URL `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`.

[5] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification.* Oracle America, Inc., Java SE 7 edition, February 2012. URL `http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`.

[6] *C# Language Specification.* Microsoft Corporation, 4.0 edition, 2010. URL `http://go.microsoft.com/fwlink/?LinkId=199552`.

[7] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications, a formal introduction.* John Wiley & Sons, July 1992. URL `http://www.daimi.au.dk/ hrn`. Gereviseerde editie van de website van july 1999.