Evolutie van Multiple Inheritance in Python

Voor Semantiek en Correctheid

Pol Van Aubel – polvanaubel@student.ru.nl – s3056414
Wouter Geraedts – w.geraedts@student.ru.nl – s0814857

25 juni 2010

Samenvatting

We tonen aan aan de hand van natuurlijke semantiek dat de mechanismen voor method resolution order in Python 2.2 ongeschikt zijn voor multiple inheritance. Hiervoor modelleren wij deze mechanismen in natuurlijke semantiek en middels wiskundige functies. Vervolgens geven we tegenvoorbeelden die aantonen dat twee belangrijke eigenschappen van multiple inheritance niet gegarandeerd worden door deze mechanismen. Uiteindelijk modelleren we het huidige C3-algoritme en demonstreren de werking aan de hand van de correcte afhandeling van de eerdere tegenvoorbeelden.

1 Inleiding

In dit paper beschrijven we het gedrag van de zogenaamde method resolution order van multiple inheritance. Dit gedrag wordt gebaseerd op de methodes die Python voor deze concepten gebruikt. Multiple inheritance is een mechanisme om een klasse van meerdere andere klassen te laten overerven. Zo is het in Python mogelijk om bijvoorbeeld de klasse “auto” van zowel “weggebruiker” als “motorvoertuig” te laten erven. Er bestaat een aantal verschillende manieren om dit af te handelen.

1.1 Objectoriëntatie

Om dit te beschrijven is het allereerst nodig om een model voor *objectgeoriënteerde* constructies als klassen, objecten en functies op te stellen. Hoewel het in de realiteit belangrijk is om data in een object op te kunnen slaan alsof het een complexe datastructuur (`struct`) is, beperken we ons tot het overerven van functies. Zodoende hoeven we geen opslag voor variabelen te definiëren. Tevens hoeft een object enkel naar de bijbehorende klasseïmplementatie te verwijzen. Ook *encapsulatie* (het beschermen van methodes en eigenschappen binnen een object) is onnodig voor dit onderzoek en is daarom niet gemodelleerd.

Enkelvoudige overerving (single inheritance) in objectgeoriënteerde programmeertalen houdt in dat een klasse één ouderklasse heeft waarvan hij de eigenschappen (attributen) overneemt. Hieronder vallen onder andere de functies en variabelen van de klasse. Een instantie van een klasse noemt men meestal een object – hoewel wij deze term ook breder gebruiken als we uitspraken doen over meerdere types (ingebouwde types in Python, door

gebruiker gedefinieerde klassen, functies). We beschrijven verderop in dit document op welke manier een object naar zijn klassedefinitie verwijst.

Meervoudige overerving (multiple inheritance) houdt daarentegen in dat een klasse meerdere ouderklassen kan hebben, waarvan hij eigenschappen overerft. Als hierbij een eigenschap conflicteert (functiedefinities van dezelfde functie in meerdere ouderklassen, variabelen met dezelfde naam en verschillende types of waardes) moeten er regels zijn waarmee bepaald kan worden van welke ouderklasse de eigenschap wordt overgenomen. Deze regels heten de MRO-regels. Dit mechanisme wordt verderop in het document beschreven.

1.2 Gebruikte semantische technieken

Voor het beschrijven van de semantiek van Python wordt de `While`-taal in natuurlijke semantiek gebruikt en uitgebreid. We baseren ons op de semantiekregels en syntaxbeschrijving van pagina's 19–32 en tabel 2.1 van [4].

Voor de beschrijving van de MRO-regels gebruiken we meestal wiskundige functies. Echter, waar dit te complex wordt hebben we het imperatieve algoritme behouden en doorlopen we dit stapsgewijs om tot het resultaat te komen.

1.3 Uitgevoerde analyse

Dit onderzoek is gestart om het gedrag van late binding in combinatie met multiple inheritance te onderzoeken. Het modelleren van multiple inheritance bleek echter zeer interessant en relatief complex, zodoende is de analyse van late binding amper uitgevoerd. Tevens zijn we tot de conclusie gekomen dat het gedrag van late binding amper afhangt van de gebruikte method resolution order, mede daarom hebben we onze focus verlegd naar multiple inheritance.

Omdat er meerdere manieren zijn om multiple inheritance af te handelen hebben we eerst onderzocht welke van deze manieren in Python gebruikt worden. Python heeft in de laatste 10 jaar drie verschillende algoritmes gebruikt om de method resolution order te bepalen. Twee van deze algoritmes zijn ongeschikt gebleken, in dit paper wordt ook uitgelegd aan de hand van de semantiek waarom dit zo is.

2 Python "Oude Stijl"

In het eerste deel van dit paper analyseren we het potentiële gedrag van multiple inheritance in Python "oude stijl". Hiermee worden implementaties van Python van voor versie 2.2 bedoeld. Deze versies hebben significante verschillen in de manier waarop ingebouwde types en door de programmeur gedefinieerde klassen werken.

Met de introductie van Python 2.2 werden deze verschillen verminderd. Python "oude stijl" is interessant omdat hier voor de zogenaamde method resolution order, de manier waarop bepaald wordt welke methode van welke klasse wordt aangeroepen, een vorm van depth-first-search gebruikt wordt. Dit algoritme is vervangen door een complexer algoritme in Python 2.2, wat op zijn beurt vervangen is door het zogenaamde C3-algoritme in Python 2.3. De reden voor deze vervanging is dat het "oude stijl"-algoritme niet voldeed; wij zullen in de volgende secties aantonen waarom.

2.1 Python's klassemodel

In versies van Python voor versie 2.2 waren er significante verschillen tussen de "types" die Python zelf bevat en "klassen" die door de gebruiker gedefinieerd worden. Deze verschillen waren een doorn in het oog van de ontwikkelaars [11]. Zodoende werd in Python 2.2 een poging gedaan deze verschillen recht te trekken. Deze poging leidde tot een eenduidige Application Programming Interface (API) voor klassen en types. Dit ging gepaard met de introductie van "nieuwe stijl"-klassen. Deze klassen erven uiteindelijk van het ingebouwde type "object". Onze interesse gaat vooral uit naar dit type klassen. Omdat types tegenwoordig dezelfde API gebruiken kunnen we ze veilig buiten beschouwing laten of behandelen als klassen.

In een klassehiërarchie met multiple inheritance moet de volgorde worden bepaald van welke methodes andere methodes overschrijven. Deze volgorde kunnen we beschrijven middels een lijst van klassen, waarbij klassen die eerder in de lijst voorkomen de methodes van klassen die later voorkomen overschrijven. Deze lijst noemen we de Method Resolution Order *van de klasse* (klasse-MRO). De regels op basis waarvan deze lijst wordt opgebouwd heten de Method Resolution Order-*regels* (MRO-regels).

Python gebruikt late binding om attributen van objecten op te vragen. De taal implementeert dit middels lijsten die door de runtime en vanuit het programma zelf geraadpleegd kunnen worden. De methodes van een object x kunnen gevonden worden door ze te zoeken in x.__class__. Dit attribuut bevat een klassebeschrijving in de vorm van een lijst, __dict__, waarin de functies die deze klasse definieert staan. Tevens bevat het attribuut x.__class__ een geordende lijst __bases__. Deze lijst bevat de klassen waarop de klasse van x gebaseerd is, de ouderklassen. [11] [12]

Deze lijsten worden gebruikt om te bepalen welke methode er moet worden aangeroepen. In Pythonversies voor versie 2.2 (Python-pre2.2) worden hierbij twee simpele regels gebruikt:

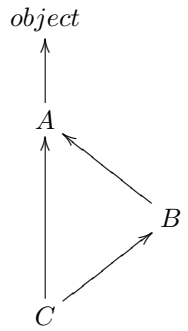
1. Attributen in een subklasse overschrijven attributen in een ouderklasse
2. Attributen in een ouderklasse overschijven attributen van ouderklassen die later in de lijst __bases__ voorkomen

Deze twee regels samen worden ook wel de "left-to-right, depth-first"-regel genoemd. Er bestaat nog een derde regel die stelt dat attributen in een instantie van een klasse de attributen van de klasse zelf overschrijven, maar aangezien we geen attributen in instanties modelleren kunnen we deze regel negeren. [11]

We kunnen __bases__ gebruiken om een gerichte graaf te maken die de klassehiërarchie beschrijft. Python legt een restrictie op aan de programmeur, namelijk dat deze graaf acyclisch moet zijn. Het kan zodoende niet voorkomen dat een klasse A een subklasse van B is, die op zijn beurt weer (eventueel via een omweg) een subklasse van A is. Voor de code

```
class A(object): pass
class B(A): pass
class C(A,B): pass
```

ziet deze graaf eruit als figuur 1.



Figuur 1: Een voorbeeld van een klassehiërarchie

We zullen nu semantische en syntactische regels toevoegen aan de taal `While` die dit modelleren. Vervolgens gebruiken we deze uitgebreide semantiek om aan te tonen dat de "left-to-right, depth-first"-regel niet voldoet als MRO-regel voor het nieuwe klassemechanisme geïntroduceerd in Python 2.2.

2.2 Uitbreiding van While

We gebruiken de syntax zoals gedefinieerd in hoofdstuk 1 van [4] en de natuurlijke semantiek zoals gedefinieerd in hoofdstuk 2 van [4] van de taal `While`. De basisregels voor syntax en semantiek zijn terug te vinden op pagina's 7–17. De semantische regels voor de statements uit `While` staan in hoofdstuk 2.1 en voornamelijk in tabel 2.1.

2.2.1 Syntax

Definiëren functies We hebben syntax nodig van functiedeclaraties, middels het func-statement:

$$D_F \quad ::= \quad \texttt{func}\ f\ \texttt{is}\ S\ \texttt{end}\ D_F \mid \varepsilon$$

waarbij f een meta-variabele is over de syntactische categorie $F\mathbf{name}$ van functienamen. Het is mogelijk om binnen een functie klassen te definiëren. Van deze functionaliteit maken we echter geen gebruik. ε is de "lege" waarde voor D_F. Zodoende kan D_F gebruikt worden om meerdere functies achtereenvolgens te definiëren.

Definiëren klassen Om klassen te definiëren hebben we de syntax nodig van het class-statement. Deze wordt

$$\texttt{class}\ c\ D_C\ D_F\ \texttt{end}$$

waarbij c een meta-variabele is over de syntactische categorie $C\mathbf{name}$ van klassenamen. Tevens definiëren we D_C als

$$D_C \quad ::= \quad c\texttt{;}\ D_C \mid \varepsilon$$

met c als meta-variabele over de categorie C_{name}. D_C is dus een lijst van klassenamen: dit zijn de al gedefinieerde klassen die de ouders zijn van de huidige klasse.

In deze paper worden zowel voor klassenamen als voor functienamen alleen hoofdletters, kleine letters, cijfers en de underscore (_) gebruikt, aan elkaar geschreven. F_{name} en C_{name} bevatten in principe dezelfde namen, maar de namen die daadwerkelijk in gebruik zijn in een programma mogen geen overlap vertonen.

Instantiatie Nu resten nog de mechanismes om functies aan te roepen en nieuwe objecten te instantiëren. Python gebruikt voor beide dezelfde syntax.

```
#functieaanroep
functie()
#klasse-instantiatie
instantie = klasse()
```

Bij functies wordt dit geïnterpreteerd als een functiecall, bij klassen als een instantiatie.

Wij zullen echter wel onderscheid maken. Instantiatie gebeurt door het new-statement.

`new` o `is` c

Hierbij is c een klassenaam uit C_{name} en o een meta-variabele over de syntactische categorie $O_{\textbf{name}}$ van objectnamen. Vervolgens is o een "instantie" van de klasse c. In de semantische regels wordt gedefinieerd wat dit precies inhoudt.

Functieaanroep Functieaanroepen worden gedaan middels het call-statement. De syntax hiervan is

`call` $o.f$

waarbij o een objectnaam is en f weer uit de verzameling Fname komt. Functieaanroepen hebben geen argumenten.

Overzicht De syntax is nu als volgt:

$$
\begin{array}{lcl}
S & ::= & x \texttt{ := a} \mid \texttt{skip} \mid S_1 \texttt{ ; } S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\
 & \mid & \texttt{while } b \texttt{ do } S \mid \texttt{class } c\ D_C\ D_F \texttt{ end} \mid \texttt{ new } o \texttt{ is } c \mid \texttt{call } o.f \\
D_F & ::= & \texttt{func } f \texttt{ is } S \texttt{ end } D_F \mid \varepsilon \\
D_C & ::= & c\texttt{; } D_C \mid \varepsilon
\end{array}
$$

Keywords als `class`, `end` en `while` zijn gereserveerd en mogen daarom niet voorkomen als variabele, klassenaam, objectnaam of functienaam.

2.2.2 Semantiek

Definitie van geordende lijsten Per klasse wordt er een lijst van overgeërfde klassen (de ouders) bijgehouden. Dit is een geordende lijst die we dan ook eerst dienen te definiëren. Dit doen we inductief, gebruik makende van pattern matching, voor het generieke type T. Hierbij is het af te leiden dat een lijst van het type L_T ofwel een *Nil* ofwel een *Cons* is.

$$
L_T : \begin{cases} \text{Nil} \\ \text{Cons}(T, L_T) \end{cases}
$$

Figuur 2: Definitie van een geordende lijst

Om van deze constructie gebruik te kunnen maken dienen er hulpfuncties om bijvoorbeeld de voorkant van een geordende lijst op te halen gemaakt te worden. Dit zijn de volgende functies:

$$\text{hd} : L_x \hookrightarrow x$$
$$\text{hd}(\text{Nil}) = \bot$$
$$\text{hd}(\text{Cons}(x, xs)) = x$$

$$\text{tl} : L_x \hookrightarrow L_x$$
$$\text{tl}(\text{Nil}) = \bot$$
$$\text{tl}(\text{Cons}(x, xs)) = xs$$

$$\text{li} : x \hookrightarrow L_x$$
$$\text{li}(x) = \text{Cons}(x, \text{Nil})$$

$$(+) : L \times L \hookrightarrow L$$
$$(+)(\text{Nil}, y) = y$$
$$(+)(\text{Cons}(x, xs), y) = \text{Cons}(x, xs + y)$$

$$[] : L_x \times \mathbb{N} \hookrightarrow x$$
$$l[0] = \text{hd}(l)$$
$$l[n+1] = \text{tl}(l)[n]$$

Figuur 3: Hulpfuncties t.b.v. manipulatie en toegang van geordende lijsten

Bij het opstellen van deze lijst is er een mate van lezersbegrip aangenomen. Zo is ervoor gekozen om de sommatie ($\sum$) niet expliciet te definiëren. De werking van dit mechanisme zou vanzelfsprekend moeten zijn, omdat additie van geordende lijsten wel is gedefinieerd. Hetzelfde geldt bijvoorbeeld voor de universele kwantor ($\forall$) en het in-predikaat ($\in$).

Definitie van environment Voor de semantiekbeschrijving gebruiken we in totaal 3 typen environments: voor klasse-, methode- en objectdefinities.

Elke klasse heeft zijn eigen lokale methode- en klasse-environment. Tijdens uitvoering worden deze environments bovenop de bestaande environments geplaatst en gebruikt. De klasse-environment zoals hij bestaat op het moment dat een klasse gedeclareerd wordt is de environment die wordt gebruikt om de lokale environment van de klasse op te bouwen. Dit t.b.v. verwijzingen naar klassen waarvan overgeërfd wordt.

Als laatste is er een globale scope voor instanties van klassen (objecten). In de vorm van een environment wordt deze, net als de environment van de klassen, samen met de state gepropageerd.

$$\text{Env}_C : C_{\text{name}} \hookrightarrow \text{Env}_F \times \text{Env}_C \times L_C$$
$$\text{Env}_F : F_{\text{name}} \hookrightarrow \text{Stm}$$
$$\text{Env}_O : O_{\text{name}} \hookrightarrow C_{\text{name}} \times \text{Env}_C$$

Om deze functies te updaten op het moment dat een class-statement wordt aangetroffen hebben we de functies upd_C en upd_F nodig; voor Env_O is dat onnodig omdat deze direct wordt veranderd in de semantiekregel.

$\text{upd}_F : D_F \times \text{Env}_F \hookrightarrow \text{Env}_F$
$\text{upd}_F(\varepsilon, env_F) = env_F$
$\text{upd}_F(\text{func } f \text{ is } S \text{ end } d_F, env_F) = \text{upd}_F(d_F, env_F[f \mapsto S])$

$\text{upd}_C : C_{\text{name}} \times D_C \times D_F \times \text{Env}_C \hookrightarrow \text{Env}_C$
$\text{upd}_C(c, d_C, d_F, env_C) = env_C[c \mapsto (\text{upd}_F(d_F, \emptyset), env_C, \text{convert}(d_C))]$

Als laatste dient de functie convert nog gedefinieerd te worden. Deze functie zet een constructie van het type D_C om naar een lijst van klassenamen.

$\text{convert} : D_F \hookrightarrow L_C$
$\text{convert}(\varepsilon) = \text{Nil}$
$\text{convert}(c \ ; \ d_C) = \text{Cons}(c, \text{convert}(d_C))$

Definitie van semantiekregels Voor deze toevoeging op de taal `While` moeten drie nieuwe semantiekregels gedefinieerd worden. Daarnaast dient de applicatie van semantiekregels aangepast te worden om de propagatie van environments te faciliteren. Dit houdt in dat in plaats van steeds een enkele state s er een triplet van het type $S \times \text{Env}_O \times \text{Env}_C$ doorgegeven dient te worden.

De bestaande semantiekregels veranderen nauwelijks. Ze dienen enkel de doorgifte van een eventueel veranderde env_O of env_C toe te staan. Als voorbeeld tonen we hier de aangepaste compositieregel:

$$[\text{comp}_{\text{ns}}] \quad \frac{\langle S_1, (s, env_O, env_C)\rangle \to (s', env'_O, env'_C) \quad \langle S_2, (s', env'_O, env'_C)\rangle \to (s'', env''_O, env''_C)}{\langle S_1; S_2, (s, env_O, env_C)\rangle \to (s'', env''_O, env''_C)}$$

Verder dienen er regels gedefinieerd te worden voor het aanroepen van methodes, instantiëren van klassen en het definiëren van klassen.

Let op dat bij **call** eventueel gedefinieerde klassen niet terug naar beneden gepropageerd worden. Omdat objecten in een globale scope worden bijgehouden, propageren deze wel in een boom terug naar beneden.

$$[\text{call}_{\text{ns}}] \quad \frac{\langle S, (s, env_O, env_C)\rangle \to (s', env'_O, env'_C)}{\langle \text{call } o.f, (s, env_O, env_C)\rangle \to (s', env'_O, env_C)} \text{ where } \begin{cases} (c, env'_C) = env_O(o) \\ S = \text{resolve}(env'_C, c, f) \end{cases}$$

$$[\text{new}_{\text{ns}}] \quad \langle \text{new } o \text{ is } c, (s, env_O, env_C)\rangle \to (s, env_O[o \mapsto (c, env_C)], env_C)$$

$$[\text{class}_{\text{ns}}] \quad \langle \text{class } c \ l_C \ d_F \text{ end}, (s, env_O, env_C)\rangle \to (s, env_O, \text{upd}_C(c, l_C, d_F, env_C))$$

Van deze regels is enkel de functie resolve nog niet gedefinieerd. Dit is een generieke functie die, afhankelijk van het gewenste MRO-gedrag, een andere implementatie heeft.

Modellering van MRO middels resolve De resolve-functie is afhankelijk van een generieke resolveOrd$_T$-functie die een ordening van klassen teruggeeft. Deze ordening geeft aan waar als eerste naar de gewenste implementatie van een methode onder de naam f gekeken dient de worden. Vervolgens wordt de functie findFunction gebruikt om ook daadwerkelijk de eerste implementatie van die functie f in die ordening te vinden.

Hier wordt ook een voorbeeld gegeven van een resolveOrd implementatie: de naïeve depth-first search.

$$\text{resolve} : \text{Env}_C \times C_{\text{name}} \times F_{\text{name}} \hookrightarrow \text{Stm}$$
$$\text{resolve}(env_C, c, f) = \text{findFunction}(\text{resolveOrd}_T(c, env_C), f)$$

$$\text{findFunction} : L_{\text{Env}_F} \times F_{\text{name}} \hookrightarrow \text{Stm}$$
$$\text{findFunction}(Nil, f) = \bot$$
$$\text{findFunction}(\text{Cons}(env_F, l_{env_F}), f) = \begin{cases} env_F(f) & \text{if } env_F(f) \neq \bot \\ \text{findFunction}(l_{env_F}, f) & \text{else} \end{cases}$$

$$\text{resolveOrd}_T : C_{\text{name}} \times \text{Env}_C \hookrightarrow L_{\text{Env}_F}$$
$$\text{resolveOrd}_{\text{naïef}}(c, env_C) = \text{Cons}(env_F, \sum_{c' \in l_C} \text{resolveOrd}_{\text{naïef}}(c', env'_C))$$
$$\text{where } (env_F, env'_C, l_C) = env_C(c)$$

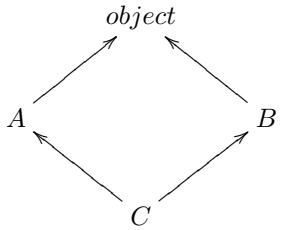
2.3 Tekortkomingen van MRO

2.3.1 Gewenste eigenschappen van multiple inheritance

We hebben eerder gesteld dat de "left-to-right, depth-first"-regel niet voldoet als MRO-regel voor Python. Dit komt doordat er twee belangrijke eigenschappen zijn die in een overervingshiërarchie bewaard moeten blijven:

1. Lokale voorrang, ook wel lokale precedentie of local precedence genoemd [3] [7]
2. Monotoniciteit [2] [10]

Lokale precedentie Stel we maken een klassehiërarchie die er uitziet als figuur 4.

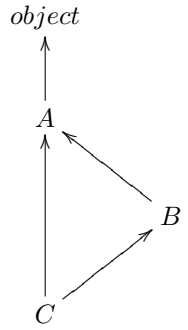


Figuur 4: Een klassieke diamantstructuur

Als de klasse C gedefinieerd is als `class C(A,B)` dan is de intuïtie dat de klasse A vóór klasse B voorkomt in de klasse-MRO van C. Als klasse C daarentegen gedefinieerd is als `class C(B,A)` dan zou B voor klasse A moeten voorkomen. Dit is lokale precedentie.

Monotoniciteit Monotoniciteit is, kort gezegd, de eigenschap dat als klasse X voorkomt voor klasse Y in de klasse-MRO van Z, dat dan klasse X ook voorkomt voor klasse Y in de klasse-MRO van klassen die Z als ouder hebben.

Een simpel voorbeeld kunnen we beschrijven aan de hand van figuur 5.



Figuur 5: Directe overerving van een klasse en zijn directe ouder

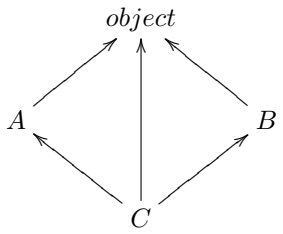
De klasse-MRO van B is (*B, A, object*). Aangezien C erft van B zou in de klasse-MRO van C eerst B, vervolgens A en dan pas object mogen voorkomen. De verwachte klasse-MRO is dan ook (*C, B, A, object*). Let wel dat er tussen B en A gerust andere klassen zouden mogen worden ingevoerd middels de MRO-regels, zolang de ordening van B en A onderling behouden blijft.

2.3.2 Lokale precedentie en monotoniciteit bij depth-first

We kunnen met een simpel voorbeeld aantonen dat deze regels lokale precedentie en monotoniciteit niet bewaren middels een kleine aanpassing van het klassieke diamantprobleem uit figuur 4. De volgende `While`-code definieert deze structuur.

```
1  x := 0;
2  class object
3          func test is x := 1 end
4  end;
5  class A object; end;
6  class B object;
7          func test is x := 2 end
8  end;
9  class C A;B;object; end;
10 new obj is C;
11 call obj.test
```

De graafrepresentatie van de structuur staat in figuur 6.



Figuur 6: Een aangepaste diamantstructuur

De semantische parseerboom is te zien in figuur 7. De statements worden in een groot deel van de boom afgekort als regelnummers en pas voluit geschreven als ze van belang zijn. Tevens is de parseerboom onderbroken bij het sterretje rechts. Hij gaat vervolgens meteen weer verder bij het sterretje onder de bovenste boom. De streep onder het sterretje in de onderste boom en de streep boven het sterretje in de bovenste boom zijn in principe dezelfde streep.

Er zijn een aantal afkortingen in de parseerboom gebruikt om ruimte te besparen. Deze afkortingen worden hieronder uitgevouwen.

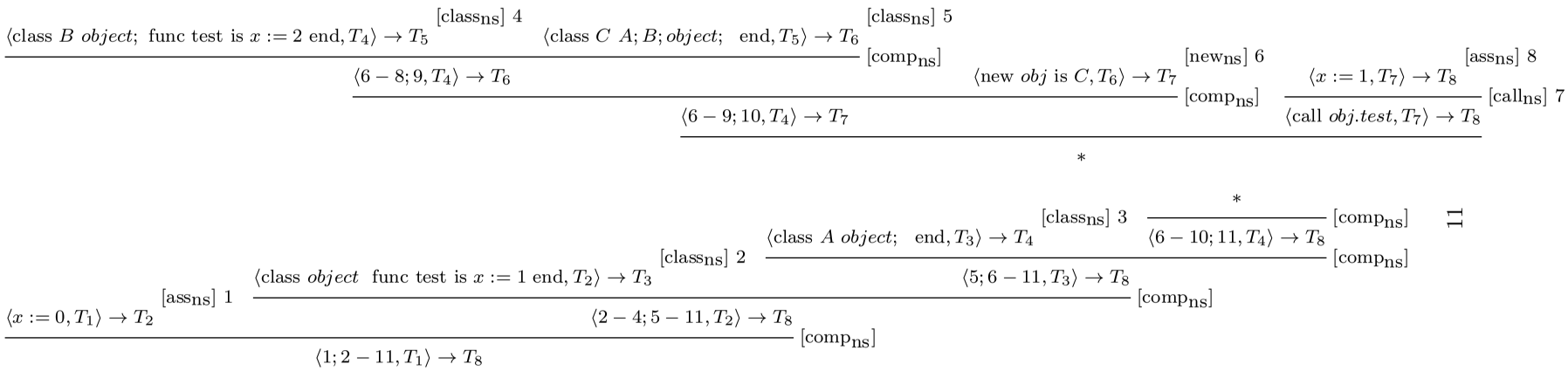
Afkorting		Betekenis
T_1	:=	$(s, \emptyset, \emptyset)$
T_2	:=	$(s_0, \emptyset, \emptyset)$
T_3	:=	$(s_0, \emptyset, env_C^1)$
T_4	:=	$(s_0, \emptyset, env_C^2)$
T_5	:=	$(s_0, \emptyset, env_C^3)$
T_6	:=	$(s_0, \emptyset, env_C^4)$
T_7	:=	(s_0, env_O^1, env_C^4)
T_8	:=	(s_1, env_O^1, env_C^4)
env_O^1	:=	$\emptyset[obj \mapsto (C, env_C^4)]$
env_C^1	:=	$\emptyset[object \mapsto (env_F^{obj}, \emptyset, \text{Nil})]$
env_C^2	:=	$env_C^1[A \mapsto (\emptyset, env_C^1, \text{Cons(object,Nil)})]$
env_C^3	:=	$env_C^2[B \mapsto (env_F^B, env_C^2, \text{Cons(object,Nil)})]$
env_C^4	:=	$env_C^3[C \mapsto (\emptyset, env_C^3, \text{Cons(A,Cons(B,Cons(object,Nil))))})]$
env_F^{obj}	:=	$\emptyset[test \mapsto x := 1]$
env_F^B	:=	$\emptyset[test \mapsto x := 2]$

Tabel 1: Afkortingen uit de semantische parseerboom in figuur 6

De T-afkortingen zijn triplets van (s, env_O, env_C). Bij de state slaat de index op de waarde van x. Als de index mist is x nog ongedefinieerd.

Hieronder volgt extra uitleg bij de toegepaste regels.

Bij regel 1 wordt een assignment uitgevoerd. Deze assignment verloopt volgens de bekende semantiekregels en verandert alleen de state naar s_0. De nieuwe triplet is afgekort als T_2.



Figuur 7: Semantische parseerboom voor code uit sectie 2.3.2

Regel 2 is de eerste toepassing van een van de nieuwe regels, [class_{ns}]. Hierbij veranderen in T_2 de state en object-environment niet, maar om tot de nieuwe class-environment te komen wordt de functie upd_C als volgt toegepast:

$$upd_C(object, \varepsilon, \text{func } test \text{ is } x := 1 \text{ end } \varepsilon, \emptyset)$$

Volgens de definitie van upd_C op pagina 7 wordt de nieuwe env_C dan

$$\emptyset[object \mapsto (upd_F(\text{func } test \text{ is } x := 1 \text{ end } \varepsilon, \emptyset), \emptyset, convert(\varepsilon))]$$

oftewel de lege functie met daarin een element dat het resultaat tussen haakjes toewijst aan *object*. Als we de toepassing van upd_F bekijken

$$upd_F(\text{func } test \text{ is } x := 1 \text{ end } \varepsilon, \emptyset)$$

vinden we eerst $upd_F(\varepsilon, \emptyset[test \mapsto x := 1])$ en vervolgens komen we tot de conclusie dat het resultaat dan $\emptyset[test \mapsto x := 1]$ moet zijn. Dit is afgekort als env_F^{obj}. $convert(\varepsilon)$ resulteert in Nil, de nieuwe class-environment wordt dus

$$\emptyset[object \mapsto (env_F^{obj}, \emptyset, \text{Nil})]$$

met als afkorting env_C^1. De nieuwe triplet wordt $(s_0, \emptyset, env_C^1)$ en deze is afgekort als T_3

Bij regel 3 gebeurt eenzelfde soort toepassing van [class_{ns}].

$$upd_C(A, object; \varepsilon, \varepsilon, env_C^1)$$

De toevoeging van env_C^1 zorgt ervoor dat de nieuwe klasse in de huidige class-environment wordt gedefinieerd. De bijbehorende nieuwe env_C wordt dan ook

$$env_C^1[A \mapsto (upd_F(\varepsilon, \emptyset), env_C^1, convert(object; \varepsilon))]$$

$upd_F(\varepsilon, \emptyset)$ is uiteraard $\emptyset$. $convert(object; \varepsilon)$ wordt een lijst Cons(object, $convert(\varepsilon)$), ofwel Cons(object, Nil). De environment wordt dus

$$env_C^1[A \mapsto (\emptyset, env_C^1, \text{Cons(object, Nil)})]$$

en deze is afgekort als env_C^2. De triplet $(s_0, \emptyset, env_C^2)$ is afgekort als T_4.

Regel 4 is een combinatie van de factoren in regel 2 en regel 3. De klassedefinitie bevat ouders en een functiedefinitie. De functiedefinitie levert uiteindelijk via

$$upd_F(\text{func } test \text{ is } x := 2 \text{ end } \varepsilon, \emptyset)$$

en

$$upd_F(\varepsilon, \emptyset[test \mapsto x := 2])$$

$\emptyset[test \mapsto x := 2]$ op. Deze is afgekort als env_F^B.

Volledige toepassing van upd_C geeft

$$env_C^2[B \mapsto (env_F^B, env_C^2, \text{Cons(object, Nil)})]$$

en deze is afgekort als env_C^3. De triplet $(s_0, \emptyset, env_C^3)$ heeft de afkorting T_5.

In regel 5 wordt de klasse met multiple inheritance gedefinieerd. Deze klasse krijgt een lege functie-environment. De enige interessante operatie is de toepassing van $convert(A; B; object; \varepsilon)$. Dit levert

$$\text{Cons(A,}convert(B; object; \varepsilon))$$

$$\text{Cons(A,Cons(B,}convert(object; \varepsilon)))$$

$$\text{Cons(A,Cons(B,Cons(object,}convert(\varepsilon))))$$

en dus uiteindelijk Cons(A,Cons(B,Cons(object,Nil))).

De env_C wordt dus

$$env_C^3[C \mapsto (\emptyset, env_C^3, \text{Cons(A,Cons(B,Cons(object,Nil))))})]$$

wat is afgekort tot env_C^4. De triplet die hieruit ontstaat is T_6.

Regel 6 is de objectinstantiatie, [new$_{\text{NS}}$]. Deze wordt toegepast op T_6, dus op $(s_0, \emptyset, env_C^4)$. Volgens de definitie van de regel worden de state en class-environment niet aangepast. De object-environment wordt direct veranderd naar $env_O[o \mapsto (c, env_C)]$. Aangezien de env_O in deze toepassing de lege verzameling is wordt dit dus

$$\emptyset[obj \mapsto (C, env_C^4)]$$

wat is afgekort tot env_O^1. De triplet (s_0, env_O^1, env_C^4) is T_7.

Regel 7 is de functieaanroep. Hier wordt de klasse-MRO van C berekend vanuit triplet T_7, dus (s_0, env_O^1, env_C^4). De MRO-regel die we momenteel gebruiken is de naïeve "left-to-right, depth-first"-regel. Om te achterhalen welk statement S er moet worden uitgevoerd wordt eerst bepaald van welke klasse het object obj een instantie is. Dit wordt gedaan door de object-environment toe te passen op de objectnaam: $env_O^1(obj)$. env_O^1 is gedefinieerd als $\emptyset[obj \mapsto (C, env_C^4)]$, dus dit levert (C, env_C^4) op. Vervolgens wordt de functie resolve aangeroepen om te bepalen welk statement er bij de functienaam test hoort:

$$resolve(env_C^4, C, test)$$

wat meteen de volgende aanroep oplevert:

$$findFunction(resolveOrd_{naïef}(C, env_C^4), test)$$

$resolveOrd_{naïef}(C, env_C^4)$ bekijkt eerst $env_C^4(C)$ om de functie-environment van C ($\emptyset$), de class-environment zoals hij was op het moment van het definiëren van C (env_C^3), en de lijst van ouders van C (Cons(A, Cons(B, Cons(object, Nil)))) te achterhalen.

Vervolgens zet hij de functie-environment van C vooraan in een nieuwe lijst. Daarachter zet hij de functie-environments van de klasse-MRO van de eerste ouder van C, zijnde A.

De klasse-MRO van A wordt op dezelfde wijze bepaald. Eerst wordt de environment zoals hij was op het moment van het definiëren van C bekeken: $env_C^3(A)$. De functie-environment die dit oplevert ($\emptyset$) wordt vooraan in een nieuwe lijst gezet. Vervolgens komen de functie-environments van de klasse-MRO van de eerste ouder van A aan bod. Deze ouder wordt uit de lijst Cons(object,Nil) gehaald. De class-environment toen A gedefinieerd werd was env_C^1.

$env^1_C(object)$ levert $(env^{obj}_F, \emptyset, \text{Nil})$. De env^{obj}_F functie-environment wordt dus vooraan een nieuwe lijst gezet. De klasse object heeft geen ouders, dus deze lijst is het resultaat van de functie: Cons(env^{obj}_F, Nil).

De klass-MRO van A wordt dan dus Cons($\emptyset$, Cons(env^{obj}_F, Nil)). Aangezien A geen andere ouders meer heeft is dit ook het resultaat waar C mee werkt.

De klasse-MRO van C is op dit moment Cons($\emptyset$, Cons($\emptyset$, Cons(env^{obj}_F, Nil))). Nu wordt de volgende ouder verwerkt.

De klasse-MRO van B wordt op dezelfde manier als die van A berekend, en dit levert Cons(env^B_F, Cons(env^{obj}_F, Nil)). De klasse-MRO van C kan nu worden uitgebreid tot Cons($\emptyset$, Cons($\emptyset$, Cons(env^{obj}_F, Cons(env^B_F, Cons(env^{obj}_F, Nil))))). Nu wordt de laatste ouder van C verwerkt: object zelf.

Het moge duidelijk zijn dat dit de klasse-MRO van C uitbreidt tot Cons($\emptyset$, Cons($\emptyset$, Cons(env^{obj}_F, Cons(env^B_F, Cons(env^{obj}_F, Cons(env^{obj}_F, Nil)))))). Dit korten we in de rest van deze analyse af tot Cons($\emptyset$, $\emptyset$, env^{obj}_F, env^B_F, env^{obj}_F, env^{obj}_F, Nil).

Als we elke env_F in deze lijst weer vervangen door de bijbehorende klassenaam krijgen we Cons(C, A, object, B, object, object, Nil).

Nu rest slechts de toepassing van $findFunction$ op de gevonden lijst en de functienaam.

$$findFunction(\text{Cons}(\emptyset, \emptyset, env^{obj}_F, env^B_F, env^{obj}_F, env^{obj}_F, \text{Nil}), test)$$

Uiteraard heeft test geen functiewaarde in de lege verzameling. Zodoende zal findFunction de derde env_F in de lijst gebruiken om $env^{obj}_F(test)$ mee te bepalen. Zoals we eerder hebben gezien is deze gedefinieerd, en wel als $test \mapsto x := 1$. Het statement S dat uitgevoerd moet worden is dus `x := 1`.

Regel 8 Het uitvoeren van het gevonden statement vergt een toepassing van een assignment-regel, [ass_{NS}]. Deze verandert alleen de state naar s_1. De nieuwe triplet is afgekort als T_8.

Conclusie Als we terugkijken naar regel 7 zien we dat de gevonden klasse-MRO van C de volgorde (C, A, object, B, object, object) heeft. De definitie van C bevat echter ouderklassen in de volgorde (A, B, object). Aangezien object na B komt in de definitie, maar voor B in de gevonden klasse-MRO, breken deze regels de lokale precedentie.

Tevens is het zo dat de klasse-MRO van B de volgorde (B, object) heeft. Men zou dus ook verwachten dat de functie `test` van B uitgevoerd zou worden, niet van object, en dat het resultaat dus zou zijn dat x uiteindelijk de waarde 2 aanneemt. Echter, in de gevonden klasse-MRO van C komt object voor B, en x neemt dan ook de waard 1 aan. Deze regels breken dus ook de monotoniciteit van multiple inheritance.

Op basis hiervan concluderen we dat ze ongeschikt zijn om dit mechanisme in de praktijk toe te passen.

3 Python 2.2

Omdat de "nieuwe stijl"-klassen sowieso niet zouden werken met het naïeve depth-first-algoritme is in Python 2.2 een nieuwe set MRO-regels ingevoerd. Er werd een tijdlang gedacht dat deze regels monotoniciteit en lokale precedentie garandeerden behalve in randgevallen waar het onmogelijk is een dusdanige MRO te maken. Een Python-gebruiker vond echter

een tegenvoorbeeld waar dit niet het geval was. [5] Op basis daarvan is onderzoek gedaan naar deze regels en geconcludeerd dat ze vervangen moesten worden.

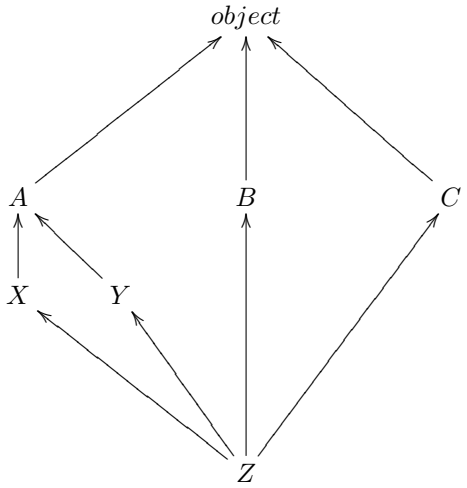
We zullen deze regels eerst beschrijven en vervolgens in natuurlijke semantiek definiëren. Daarna zullen we aantonen dat ook deze regels niet voldoen als MRO-regels voor multiple inheritance.

3.1 MRO-regels

De nieuwe regels zijn door de Python-ontwikkelaars beschreven als

1. Gebruik de MRO-regels "oude stijl" om de lijst van potentiële ouderklassen op te bouwen.
2. Verwijder, van de klassen die meerdere keren in de lijst voorkomen, alle elementen behalve de laatste. [12] [9]

Deze beschrijving klopt echter niet. De Python-code in appendix A definieert de klassehiërarchie uit figuur 8.



Figuur 8: Een klassehiërarchie waarbij de MRO zich anders gedraagt dan beschreven

De enige klasse waarbij multiple inheritance van toepassing is is *Z*. *Z* is gedefinieerd met de ouderlijst (*X, B, Y, C*).

Volgens de nieuwe MRO-regels zou eerst de lijst van ouderklassen moeten worden opgebouwd door middel van de "left-to-right, depth-first"-regel. Deze lijst ziet eruit als (*Z, X, A, object, B, object, Y, A, object, C, object*). Door volgens regel 2 de overbodige instanties te schrappen komen we tot de klasse-MRO van Z: (*Z, X, B, Y, A, C, object*). [5]

Als we echter de Python-code uit de appendix uitvoeren zien we dat Python een andere klasse-MRO eruit opbouwt, namelijk (*Z, X, Y, A, B, C, object*). De variabele `var` geeft inderdaad, in plaats van de verwachte `b`, een `y` terug. Hieruit kunnen we concluderen dat het in Python geïmplementeerde algoritme andere MRO-regels hanteert. Dit algoritme is te vinden in appendix B. Dit algoritme is volgens de schrijver "waarschijnlijk" een implementatie van het $\mathcal{L}*_{LOOPS}$-algoritme (dat hij in zijn analyse het $\mathcal{C}*_{LOOPS}$-algoritme noemt). [8] [1] [3]

Hoewel dit algoritme uitgebreid is beschreven en geanalyseerd voordat het gebruikt werd in Python blijkt het ook niet te voldoen voor multiple inheritance.

3.2 Uitbreiding van While

De syntax- en semantiekregels blijven hetzelfde als in sectie 2.2. De enige verandering is dat we de functie $\text{resolveOrd}_{22} : C_{\text{name}} \times \text{Env}_C \hookrightarrow L_{\text{Env}_F}$ niet definiëren in termen van wiskundige formules, maar als het doorlopen van het algoritme conservative merge in appendix B.

Bij de berekening van de klasse-MRO neemt Python de klasse-MRO van de eerste twee ouderklassen en voegt deze samen volgens conservative merge. Vervolgens voegt het algoritme dit resultaat samen met de klasse-MRO van de derde ouderklasse, het resultaat daarvan wordt samengevoegd met het resultaat van de vierde ouderklasse en zo verder.

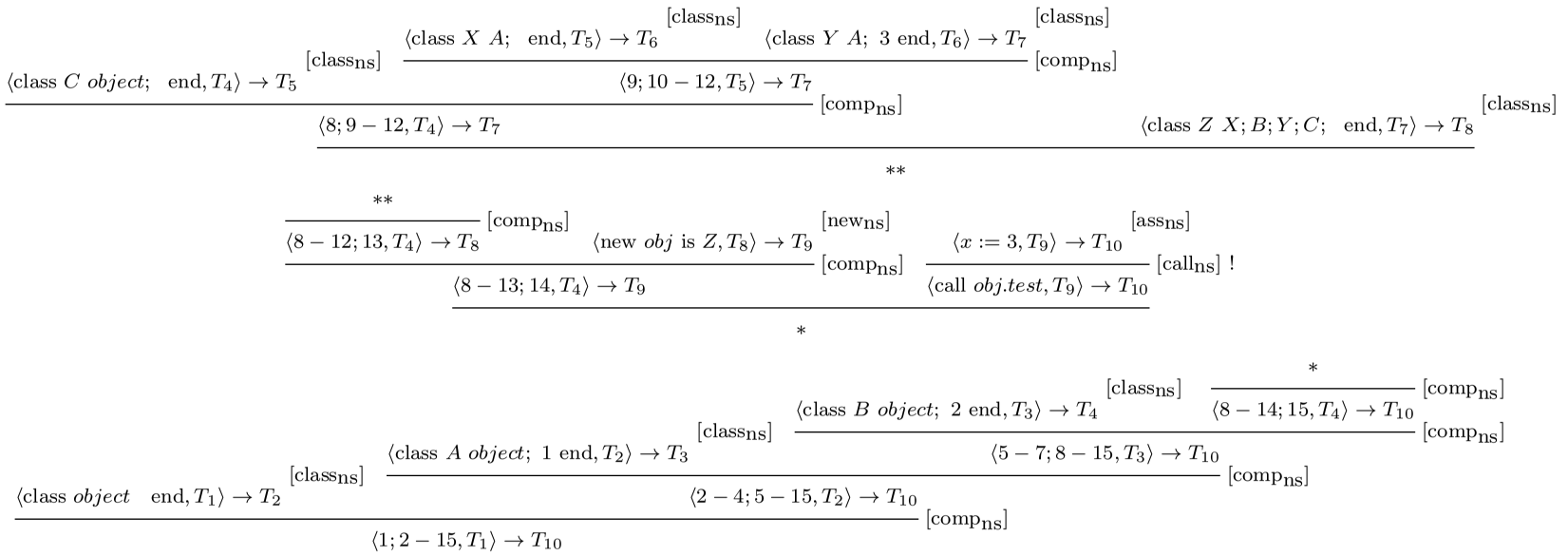
3.3 Tekortkomingen van MRO

Lokale precedentie Om de tekortkomingen van het $\mathcal{L}*_{LOOPS}$-algoritme te demonstreren gebruiken we weer de klassehiërarchie uit figuur 8.

De volgende `While`-code definieert dezelfde structuur.

```
1  class object end;
2  class A object;
3          func test is x := 1 end
4  end;
5  class B object;
6          func test is x := 2 end
7  end;
8  class C object; end;
9  class X A; end;
10 class Y A;
11         func test is x := 3 end
12 end;
13 class Z X;B;Y;C; end;
14 new obj is Z;
15 call obj.test
```

Figuur 9 bevat de semantische parseerboom voor deze code. De drie functiedefinities zijn om ruimte te besparen in deze boom afgekort tot de waarde die ze aan x toewijzen in het statement.



Figuur 9: Semantische parseerboom voor code uit sectie 3.3

Afkorting		Betekenis
T_1	:=	$(s, \emptyset, \emptyset)$
T_2	:=	$(s, \emptyset, env_C^1)$
T_3	:=	$(s, \emptyset, env_C^2)$
T_4	:=	$(s, \emptyset, env_C^3)$
T_5	:=	$(s, \emptyset, env_C^4)$
T_6	:=	$(s, \emptyset, env_C^5)$
T_7	:=	$(s, \emptyset, env_C^6)$
T_8	:=	$(s, \emptyset, env_C^7)$
T_9	:=	(s, env_O^1, env_C^7)
T_{10}	:=	(s_3, env_O^1, env_C^7)
env_O^1	:=	$\emptyset[obj \mapsto (Z, env_C^7)]$
env_C^1	:=	$\emptyset[object \mapsto (env_F^{obj}, \emptyset, \text{Nil})]$
env_C^2	:=	$env_C^1[A \mapsto (env_F^A, env_C^1, \text{Cons(object,Nil)})]$
env_C^3	:=	$env_C^2[B \mapsto (env_F^B, env_C^2, \text{Cons(object,Nil)})]$
env_C^4	:=	$env_C^3[C \mapsto (\emptyset, env_C^3, \text{Cons(object,Nil)})]$
env_C^5	:=	$env_C^4[X \mapsto (\emptyset, env_C^4, \text{Cons(A,Nil)})]$
env_C^6	:=	$env_C^5[Y \mapsto (env_F^Y, env_C^5, \text{Cons(A,Nil)})]$
env_C^7	:=	$env_C^6[Z \mapsto (\emptyset, env_C^6, \text{Cons(X,Cons(B,Cons(Y,Cons(C,Nil)))))})]$
env_F^A	:=	$\emptyset[test \mapsto x := 1]$
env_F^B	:=	$\emptyset[test \mapsto x := 2]$
env_F^Y	:=	$\emptyset[test \mapsto x := 3]$

Tabel 2: Afkortingen uit de semantische parseerboom in figuur 9

De T-afkortingen zijn triplets van (s, env_O, env_C). Bij de state slaat de index op de waarde van x. Als de index mist is x nog ongedefinieerd.

We gaan niet als in sectie 2.3.2 de volledige parseerboom semantisch analyseren, maar zullen ons beperken tot de interessante eigenschappen. De afkortingen in tabel 2 zijn correct toegewezen.

De klasse-MRO's van alle klassen behalve Z zijn in de tabel 3 in Python-notatie gegeven. Deze zijn triviaal aangezien geen enkele van deze klassen gebruik maakt van multiple inheritance. We gebruiken Python-notatie omdat het MRO-algoritme dat we gaan gebruiken in Python is geschreven. [A, B, C] komt overeen met Cons(A, Cons(B, Cons(C, Nil))) en Python-lijsten hebben een index beginnend bij 0.

Klasse		Klasse-MRO
object	$\mapsto$	[object]
A	$\mapsto$	[A, object]
B	$\mapsto$	[B, object]
C	$\mapsto$	[C, object]
X	$\mapsto$	[X, A, object]
Y	$\mapsto$	[Y, A, object]

Tabel 3: Klasse-MRO's van de single-inherited klassen uit figuur 8

De enige regel die van belang is is de [call_{ns}], gemarkeerd met een "!".

Deze call wordt uitgevoerd op triplet T_9. Het object herleidt naar klasse Z in class-environment env_C^7.

$$resolve(env_C^7, Z, test)$$

wordt

$$findFunction(resolveOrd_{22}(Z, env_C^7), test)$$

$resolveOrd_{22}(Z, env_C^7)$ komt volgens de algoritmebeschrijving neer op de volgende aanroep (met cm als afkorting voor conservative_merge en x-MRO als afkorting voor de klasse-MRO van klasse x):

`Z + cm( cm( cm(X-MRO, B-MRO), Y-MRO), C-MRO)`

ofwel

`Z + cm( cm( cm([X, A, object], [B, object]), [Y, A, object]), [C, object])`

We zullen eerst het algoritme uit appendix B stapsgewijs doorlopen voor cm([X, A, object], [B, object]).

left_size is 3, right_size is 2. In de for-loops op regels 6–11 wordt elk element van de rechterlijst eerst vergeleken met het eerste element van de linkerlijst. Daar is geen overeenkomst, dus wordt elk element van de rechterlijst vergeleken met het tweede element van de linkerlijst. Ook daar is weer geen overeenkomst. De eerste overeenkomst die het algoritme vindt is als hij het tweede element van de rechterlijst ($j = 1$) vergelijkt met het derde element van de linkerlijst ($i = 2$).

Het algoritme springt nu met deze informatie naar regel 12. Aangezien eq net op 1 is gezet gaat het de inhoud van het if-statement uitvoeren. In de for-loop van regel 14–17 wordt voor elk element van de rechterlijst met een lagere index dan j gekeken of het zich al ergens in de linkerlijst bevindt, eventueel ook na index i. Zo niet, dan wordt het element toegevoegd aan een nieuwe tijdelijke lijst die deze elementen in dezelfde volgorde bewaart. Als het element zich wel al in de linkerlijst bevindt wordt het genegeerd.

In onze situatie zal de tijdelijke lijst alleen B bevatten.

Als alle elementen tot aan j verwerkt zijn wordt deze tijdelijke lijst tussen het element op positie i en het element ervoor ingevoerd. Intuïtief blijft hierdoor monotoniciteit in deze lijsten behouden, omdat alle elementen die ingevoerd worden zich ook in de originele lijst voor het element op positie j bevonden, en het element op positie j is gelijk aan het element op positie i. We zullen ook laten zien dat dit niet het geval is.

In dit voorbeeld is de linkerlijst nu [X, A, B, object].

Vervolgens worden de elementen uit de rechterlijst die tot nu toe gecontroleerd zijn, tot en met het element op positie j, gewist. Als de rechterlijst nu leeg is is het algoritme meteen klaar. Zo niet dan wordt teruggesprongen naar de for-loops op regels 6–11 om te controleren of er nog meer overlappende elementen zijn. Als die niet meer gevonden worden dan wordt de rest van de rechterlijst achteraan de linkerlijst toegevoegd en is het algoritme klaar.

In de huidige situatie is de rechterlijst nu leeg.

De volgende stap is de aanroep van `cm` op dit resultaat en [Y, A, object]: `cm([X, A, B, object], [Y, A, object])`.

De eerste overlap zal gevonden worden bij A, als zowel index i als index j gelijk zijn aan 1. Y bevindt zich nog niet in de linkerlijst, dus wordt aan de tijdelijke lijst toegevoegd. Alle elementen met een index lager dan j zijn dan verwerkt en de tijdelijke lijst wordt ingevoerd tussen X en A in de linkerlijst: [X, Y, A, B, object].

De rechterlijst bevat nu na het wissen van Y en A alleen nog object. Dit overlapt met de linkerlijst op inder $i = 3$ en $j = 0$. Er zijn geen

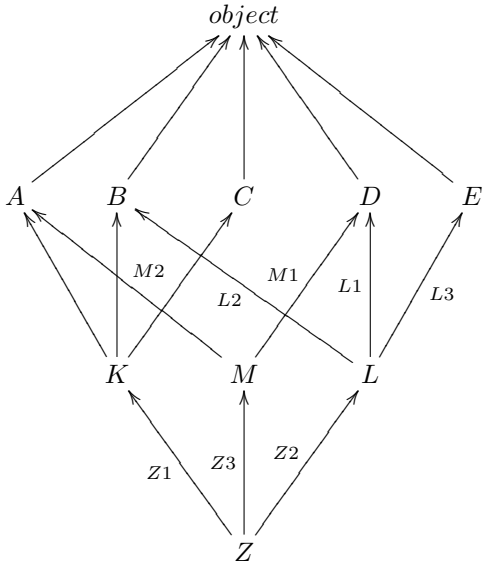
elementen met een lagere index dan 0, dus er wordt geen element in de linkerlijst ingevoegd. object wordt gewist uit de rechterlijst, die nu leeg is. Het algoritme is dus klaar.

De laatste stap is de aanroep `cm([X, Y, A, B, object], [C, object])`. Analoog aan de eerste stap is de eerste en enige overlap bij object. C wordt tussen B en object ingevoerd en na het wissen is de rechterlijst leeg. Het algoritme is klaar en het resultaat is [X, Y, A, B, C, object]. Hier wordt Z nog voor aan ingevoerd en de uiteindelijke lijst is [Z, X, Y, A, B, C, object]

Zoals eerder vermeld komt dit overeen met de lijst Cons(Z, Cons(X, Cons(Y, Cons(A, Cons(B, Cons(C, Cons(object, Nil))))))). Als we deze klassen herleiden naar de bijbehorende env_F dan levert dat de lijst Cons(∅, Cons(∅, Cons(env_F^Y, Cons(env_F^A, Cons(env_F^B, Cons(∅, Cons(∅, Nil))))))). De functie $findFunction$ toegepast op deze lijst en test zal als resultaat $env_F^Y(test)$ opleveren, en dat is `x := 3`. Dit statement is ook gebruikt in de parseerboom.

Als we nu terugkijken naar de definitie van Z in de `While`-code dan zien we dat deze gegeven is als class Z X;B;Y;C; end;. De verwachte volgorde in de klasse-MRO is dus sowieso (Z, X, B, Y, C). De uiteindelijke klasse-MRO voor Z is volgens dit algoritme echter Cons(Z,Cons(X, Cons(Y, Cons(A, Cons(B, Cons(C, Cons(object, Nil))))))), dus met de volgorde (Z, X, Y, A, B, C, object). Y komt hier voor B en dit conflicteert met lokale precedentie.

Monotoniciteit Om tevens aannemelijk te maken dat het algoritme de monotoniciteit niet waarborgt moeten we een ingewikkeldere klassehiërarchie nemen. Deze is weergegeven in figuur 10. Hierbij zijn de pijlen genummerd als de lokale precedentie in een andere volgorde dan van links naar rechts loopt.



Figuur 10: Klassehiërarchie die niet monotoon is onder $\mathcal{L}*_{LOOPS}$

De `While`-code die deze hiërarchie definieert is hieronder gegeven.

```
1  class A object; end;
2  class B object; end;
3  class C object; end;
4  class D object; end;
5  class E object; end;
6  class K A;B;C;  end;
7  class L D;B;E;  end;
8  class M D;A;    end;
9  class Z K;L;M;  end
```

De klasse-MRO's van deze hiërarchie voor alle klassen behalve Z staan in tabel 4.

Klasse		Klasse-MRO
object	$\mapsto$	[object]
A	$\mapsto$	[A, object]
B	$\mapsto$	[B, object]
C	$\mapsto$	[C, object]
D	$\mapsto$	[D, object]
E	$\mapsto$	[E, object]
K	$\mapsto$	[K, A, B, C, object]
L	$\mapsto$	[L, D, B, E, object]
M	$\mapsto$	[M, D, A, object]

Tabel 4: Klasse-MRO's van de klassen uit figuur 10

De conservative_merge van K en L levert [K, A, L, D, B, C, E, object]. De conservative_merge van dit resultaat met M levert vervolgens [K, M, A, L, D, B, C, E, object]. Z eraan toegevoegd levert [Z, K, M, A, L, D, B, C, E, object].

We zien dat M een ouder is van Z, en dat in de klasse-MRO van M de klasse A na de klasse D komt. In de uitgerekende klasse-MRO van Z komt de klasse A echter voor de klasse D. Zodoende breekt dit de monotoniciteit van multiple inheritance.

Conclusie Ook het $\mathcal{L}*_{LOOPS}$-algoritme voldoet niet voor multiple inheritance omdat het klasse-MRO's kan genereren die niet monotoon zijn of lokale precedentie niet respecteren.

4 Python 2.3

Zoals we hebben aangetoond zijn de algoritmes van Python in versies voor 2.3 eigenlijk niet geschikt voor multiple inheritance omdat ze de belangrijke eigenschappen (monotoniciteit en lokale voorrang) niet kunnen garanderen. In Python 2.3 is wegens de tekortkomingen van de eerdere regels het algoritme ingevoerd dat vervolgens in alle versies van Python gebruikt is. Dit algoritme, het zogenaamde C3-algoritme [1], zorgt er wel voor dat deze eigenschappen gewaarborgd zijn. Als voor een klassehiërarchie geen klasse-MRO bestaat waar deze eigenschappen behouden blijven faalt het C3-algoritme en geeft Python een foutmelding.

In deze sectie geven we een beschrijving van het C3-algoritme. Vervolgens introduceren we een semantische functie die het algoritme modelleert

en demonstreren we de werking aan de hand van enkele voorbeelden uit de voorgaande onderdelen.

4.1 Het C3-algoritme

Het C3-algoritme werkt recursief, door alle klasse-MRO's van alle ouderklassen coöperatief samen te voegen.

De klasse-MRO van een klasse zonder ouders is simpelweg de lijst met alleen die klasse. De klasse-MRO van object is Cons(object,Nil).

Stel echter dat een klasse X n ouders heeft, genoteerd als O_i. Elk van deze ouders heeft een klasse-MRO, die we kunnen aanduiden met MRO_i. C3(X) is dan li(X) + mergeC3(MRO_1, MRO_2, ..., MRO_n, Cons(O_1, Cons(O_2, ... Cons(O_n, Nil) ...))).

Dit is gelijk aan li(X) + mergeC3($C3(O_1)$, $C3(O_2)$, ..., $C3(O_n)$, Cons(O_1, Cons(O_2, ... Cons(O_n, Nil) ...))).

De lijsten die aan mergeC3 worden meegegeven zijn dus zowel de klasse-MRO's van de ouders als de lijst van ouders zelf. Dit laatste zorgt ervoor dat lokale precedentie behouden blijft.

MergeC3 neemt een willekeurig aantal lijsten en loopt deze lijsten in de volgorde waarin ze als argument zijn meegegeven door. Eerst bekijkt het het eerste element van de eerste lijst. Als dat in geen enkele van de tails (alle elementen behalve het eerste) van de lijsten voorkomt dan wordt het element aan de achterkant van de klasse-MRO die opgebouwd wordt toegevoegd. Vervolgens wordt het element uit alle lijsten verwijderd (het kan immers wel in andere lijsten als head (voorkant) voorkomen) en wordt mergeC3 op die aangepaste lijsten uitgevoerd.

Als het eerste element van de eerste lijst wel in de tail van een andere lijst voorkomt, dan moet er gekeken worden naar het eerste element van de tweede lijst. Voldoet dat aan de voorwaarde dat het in geen enkele tail voorkomt (ook niet van de eerste lijst) dan wordt dat element toegevoegd aan de klasse-MRO en verwijderd uit de lijsten. Vervolgens begint mergeC3 weer bij de eerste lijst.

Voldoet ook dat element niet aan de eis, dan wordt naar de head van de derde lijst gekeken, daarna naar de vierde etc. Als alle lijsten leeg zijn is het algoritme klaar en is de klasse-MRO het resultaat. Als er geen enkele head aan de voorwaarde voldoet dan is het onmogelijk om een klasse-MRO te construeren die monotoon is en lokale precedentie respecteert. Het algoritme faalt dan. In het geval van Python wordt een foutmelding gegenereerd.

4.2 Uitbreiding van While

Om mergeC3 te modelleren gebruiken we de minimalisatie-operatie uit de μ-recursie. Deze operator, met de syntax $\mu\ i$: *voorwaarde*, zoekt het kleinste getal i waarvoor een bepaalde voorwaarde geldt. Als deze voorwaarde voor geen enkele i geldt dan is de functie ongedefinieerd.

We passen de semantiek zoals hij is gedefinieerd in sectie 2.2 aan op het punt dat de functie resolve nu gebruik zal maken van $\text{resolveOrd}_{\text{C3}}$. Tevens definiëren we een aantal nieuwe functies die het werk van $\text{resolveOrd}_{\text{C3}}$ mogelijk maken.

$\text{resolveOrd}_T : C_{\text{name}} \times \text{Env}_C \hookrightarrow L_{\text{Env}_F}$
$\text{resolveOrd}_{\text{c3}}(c, env_C) = \text{mapToFunctions}(W(c, env_C))$

$M_{\text{c3}} : L_{L_{(C_{\text{name}}, \text{Env}_C)}} \hookrightarrow L_{(C_{\text{name}}, \text{Env}_C)}$
$M_{\text{c3}}(l) = \begin{cases} \text{Nil} & \text{if } l = \text{Cons}(\text{Nil}, \text{Nil}) \\ M'_{\text{c3}}(l) & \text{else} \end{cases}$
$M'_{\text{c3}}(l) = \text{hd}(l[i]) + M_{\text{c3}}(l \setminus \text{hd}(l[i])) \text{ with } i = \mu\, i : \forall_{x \in l}(\text{hd}(l[i]) \notin \text{tl}(x))$

$W_{\text{c3}} : C_{\text{name}} \times \text{Env}_C \hookrightarrow L_{(C_{\text{name}}, \text{Env}_C)}$
$W_{\text{c3}}(c, env_C) = \text{li}((c, env_C)) + M_{\text{c3}}(\sum_{x \in l} \text{li}(W_{\text{c3}}(x, env'_C)) + \text{li}(\sum_{x \in l}(x, env'_C))$
where $(env'_F, env'_C, l) = env_C(c)$

$\text{mapToFunctions} : L_{(C_{\text{name}}, \text{Env}_C)} \hookrightarrow L_{\text{Env}_F}$
$\text{mapToFunctions}(\text{Nil}) = \text{Nil}$
$\text{mapToFunctions}(\text{Cons}((c, env_C), xs)) = \text{Cons}(env'_F, \text{mapToFunctions}(xs))$
where $(env'_F, env'_C, l) = env_C(c)$

Om te voorkomen dat we per ongeluk twee klassen voor dezelfde klasse aanzien terwijl het eigenlijk verschillende klassen zijn (stel bijvoorbeeld de klasse A en B die allebei slechts één functie definiëren met exact dezelfde naam en dezelfde inhoud) kunnen we de klasse-MRO niet alleen maar een lijst van functie-environments laten zijn. In de functie W_{c3} moeten de klasse-MRO's tupels zijn van klassenamen en klasse-environments. Daarom hebben we een functie mapToFunctions die de uiteindelijke klasse-MRO van W_{c3} omzet naar een klasse-MRO van Env_F's voor gebruik door de functie findFunction. Het enige wat mapToFunctions doet is voor elk tupel uit de klasse-MRO bepalen welke Env_F erbij hoort en daar een nieuwe lijst van opbouwen.

W_{c3} is de functie die, gegeven een klasse en een klasse-environment, de klasse-MRO opbouwt. In de sectie 4.1 is deze aangeduid als de functie "C3".

M_{c3} is de functie "mergeC3" uit sectie 4.1. Deze functie neemt een lijst van lijsten en zoekt daarin de eerste lijst waarvoor geldt dat de head van die lijst niet voorkomt in de tails van alle andere lijsten. Als deze lijst niet bestaat dan is de functie ongedefinieerd en is het onmogelijk een klasse-MRO te berekenen. Als deze lijst wel bestaat dan neemt deze functie daar de head van en zet hem voor de volgende functieaanroep in een nieuwe lijst. De volgende functieaanroep is de aanroep van M_{c3} op de lijst van lijsten waaruit alle instanties van het element dat net verwerkt is zijn weggelaten.

4.3 Demonstratie van MRO

In deze sectie zullen we de werking van het C3-algoritme aan de hand van een aantal voorbeelden demonstreren.

Onberekenbaar We nemen nogmaals de klassehiërarchie uit figuur 1. De bijbehorende `While`-code is

```
1 class object end;
```

```
2  class A object; end;
3  class B A; end;
4  class C A;B; end;
```

De klasse-MRO's van object, A en B zijn triviaal te bepalen. De klasse-MRO van object is Cons(object, Nil). De klasse-MRO van A is Cons((A, env_C^A), Cons((object, env_C^{object}), Nil)) en de klasse-MRO van B is Cons((B, env_C^B), Cons((A, env_C^A), Cons((object, env_C^{object}), Nil))). De ouderlijst van C is Cons((A, env_C^A), Cons((B, env_C^B), Nil)). Ter bevordering van de leesbaarheid korten we dat nu af naar [object], [A, object], [B, A, object] en [A, B] respectievelijk. Als we nu proberen een klasse-MRO van C te berekenen middels een aanroep naar $W_{\mathrm{C3}}(C, env_C^C)$ dan levert dat

$$[C] + M_{\mathrm{C3}}([[A, object], [B, A, object], [A, B]])$$

en volgens de definitie van M_{C3} is dit de eerste head van een lijst die niet in de tail van een van de lijsten zit.

Echter, er zijn slechts twee verschillende heads beschikbaar: A en B. Beide zitten in de tail van een lijst. Het is dus onmogelijk om deze klasse-MRO te bepalen zonder ofwel lokale precedentie ofwel monotoniciteit te breken.

Klassieke diamant De klassehiërarchie uit figuur 4 is de klassieke diamantstructuur. Als hij gedefinieerd wordt door de `While`-code

```
1  class object end;
2  class A object; end;
3  class B object; end;
4  class C A;B; end;
```

dan is de klasse-MRO van object de lijst [object]. De klasse-MRO van A is [A, object], die van B is [B, object]. Als we nu proberen een klasse-MRO van C te berekenen middels een aanroep naar $W_{\mathrm{C3}}(C, env_C^C)$ dan levert dat

$$[C] + M_{\mathrm{C3}}([[A, object], [B, object], [A, B]])$$

De eerste head van een lijst die niet in de tail van een van de lijsten zit is de head van de eerste lijst, A. Dit levert dus

$$[C] + [A] + M_{\mathrm{C3}}([[object], [B, object], [B]])$$

Nu is de eerste beschikbare head B, dus wordt het

$$[C] + [A] + [B] + M_{\mathrm{C3}}([[object], [object]])$$

en de laatste beschikbare klasse is object.

[C, A, B, object] ofwel Cons((C, env_C^C), Cons((A, env_C^A), Cons((B, env_C^B), Cons((object, env_C^{object}), Nil)))).

Precedentie In sectie 3.3 hebben we aan de hand van figuur 8 en een semantische parseerboom aangetoond dat de regels van Python 2.2 de lokale precedentie niet respecteren. C3 rekent hier echter wel een correcte klasse-MRO uit.

De klasse-MRO's in tabel 3 en de afkortingen uit tabel 2 gebruiken we om een aanroep naar $W_{\mathrm{C3}}(Z, env_C^7)$ op te stellen:

$$[Z]+M_{\mathrm{C3}}([[X, A, object], [B, object], [Y, A, object], [C, object], [X, B, Y, C]])$$

$[Z]+[X]+M_{\mathrm{C3}}([[A, object], [B, object], [Y, A, object], [C, object], [B, Y, C]])$

$[Z]+[X]+[B]+M_{\mathrm{C3}}([[A, object], [object], [Y, A, object], [C, object], [Y, C]])$

$[Z]+[X]+[B]+[Y]+M_{\mathrm{C3}}([[A, object], [object], [A, object], [C, object], [C]])$

$[Z]+[X]+[B]+[Y]+[A]+M_{\mathrm{C3}}([[object], [object], [object], [C, object], [C]])$

$[Z]+[X]+[B]+[Y]+[A]+[C]+M_{\mathrm{C3}}([[object], [object], [object], [object]])$

Dus de uiteindelijke klasse-MRO is [Z, X, B, Y, A, C, object]. Dit is monotoon en behoudt lokale precedentie.

Monotoniciteit In sectie 3.3 hebben we tevens aan de hand van figuur 10 en bijbehorende code aangetoond dat de regels van Python 2.2 ook monotoniciteit niet respecteren. C3 rekent hier echter wel een correcte klasse-MRO uit.

De klasse-MRO's in tabel 4 gebruiken we om een aanroep naar $W_{\mathrm{C3}}(Z, env_C^Z)$ op te stellen:

$[Z]+M_{\mathrm{C3}}([[K, A, B, C, object], [L, D, B, E, object], [M, D, A, object], [K, L, M]])$

$[Z]+[K]+M_{\mathrm{C3}}([[A, B, C, object], [L, D, B, E, object], [M, D, A, object], [L, M]])$

$[Z]+[K]+[L]+M_{\mathrm{C3}}([[A, B, C, object], [D, B, E, object], [M, D, A, object], [M]])$

$[Z]+[K]+[L]+[M]+M_{\mathrm{C3}}([[A, B, C, object], [D, B, E, object], [D, A, object]])$

$[Z]+[K]+[L]+[M]+[D]+M_{\mathrm{C3}}([[A, B, C, object], [B, E, object], [A, object]])$

$[Z]+[K]+[L]+[M]+[D]+[A]+M_{\mathrm{C3}}([[B, C, object], [B, E, object], [object]])$

$[Z]+[K]+[L]+[M]+[D]+[A]+[B]+M_{\mathrm{C3}}([[C, object], [E, object], [object]])$

$[Z]+[K]+[L]+[M]+[D]+[A]+[B]+[C]+M_{\mathrm{C3}}([[object], [E, object], [object]])$

$[Z]+[K]+[L]+[M]+[D]+[A]+[B]+[C]+[E]+M_{\mathrm{C3}}([[object], [object], [object]])$

[Z, K, L, M, D, A, B, C, E, object] is de klasse-MRO van Z. Ook dit voldoet weer aan de voorwaarden van multiple inheritance.

Conclusie Uit ons onderzoek blijkt dat C3 de gewenste eigenschappen heeft om multiple inheritance mogelijk te maken. Het berekent een aantal twijfelgevallen goed, en als een klasse-MRO onmogelijk te construeren is eindigt het algoritme daadwerkelijk – het blijft niet oneindig loopen. Tevens is het relatief makkelijk wiskundig te definiëren.

5 Conclusie

Er is een probleem met multiple inheritance aangetoond in Python 2.2 en ervoor. We hebben onderzocht wat dit probleem precies inhoudt en een semantische analyse gedaan van de problematische cases. Onze conclusie is dat zowel de "oude stijl"-regels als de regels van Python 2.2 ongeschikt zijn voor implementatie van multiple inheritance.

Voor zover wij hebben kunnen achterhalen garandeert het C3-algoritme dat als er een mogelijke klasse-MRO met behoudt van monotoniciteit en lokale precedentie bestaat, dit algoritme hem vindt, en als deze niet bestaat dan zal het algoritme ook geen klasse-MRO opleveren. Wij verwachten dan ook niet dat dit algoritme vervangen hoeft te worden, tenzij de eisen aan multiple inheritance in Python wijzigen.

5.1 Vervolgonderzoek

Zoals in de inleiding al is aangegeven zijn we niet toegekomen aan het onderzoeken van late binding. Multiple inheritance maakt echter dingen als coöperatieve super-calls mogelijk, waarbij een class automatisch de correcte supermethodes van alle ouderklasses aanroept zonder conflicten. Dit is een zeer interessant concept en de wijze van implementatie in Python duidt op relevantie voor late binding.

Appendices

A Definitie van klassehiërarchie in Python

Deze code genereert de klassehiërarchie beschreven in sectie 3.1 op pagina 15.

```
1  var = "undef"
2
3  class A(object):
4          def test(self):
5                  global var
6                  var = "a"
7
8  class B(object):
9          def test(self):
10                 global var
11                 var = "b"
12
13 class C(object):
14         pass
15
16 class X(A):
17         pass
18
19 class Y(A):
20         def test(self):
21                 global var
22                 var = "y"
23
24 class Z(X,B,Y,C):
25         pass
26
27 print(Z.__mro__)
28
29 cls = A()
30 cls.test()
31 print var
```

B Merge-stap van Python 2.2 MRO

Deze code demonstreert de MRO-berekening door Python 2.2 voor de klassehiërarchie uit appendix A, behandeld in sectie 3.1 op pagina 15. De code is gebaseerd op code gegeven in [6].

```
1  def conservative_merge(left,right):
2      while 1:
3          left_size = len(left)
4          right_size = len(right)
5          eq = 0
6          for i in range(left_size):
7              for j in range(right_size):
8                  if left[i] == right[j]:
9                      eq = 1
10                     break
11             if eq: break
12         if eq:
13             temp = []
14             for r in range(j):
15                 rr = right[r]
16                 if rr not in left:
17                     temp.append(rr)
18             left[i:i] = temp
19             right[0:j+1] = []
20             continue
21         break
22     left.extend(right)
23     return left
24
25 x_mro = ["x", "a", "object"]
26 b_mro = ["b", "object"]
27 y_mro = ["y", "a", "object"]
28 c_mxo = ["c", "object"]
29
30 cm = conservative_merge
31
32 print(z + cm(cm(cm(list1, list2),list3),list4))
```

Referenties

[1] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A monotonic superclass linearization for dylan. `http://www.webcom.com/haahr/dylan/linearization-oopsla96.html`, June 1996. Artikel bezocht in mei 2010.

[2] R. Ducournau, M. Habib, M. Huchard, and M. L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. *SIGPLAN Not.*, 27(10):16–24, 1992.

[3] R. Ducournau, M. Habib, M. Huchard, and M. L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 164–175, New York, NY, USA, 1994. ACM.

[4] H. R. Nielson and F. Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1999. Revised Edition.

[5] S. Pedroni. [python-dev] perplexed by mro. `http://mail.python.org/pipermail/python-dev/2002-October/029035.html`, Oct. 2002. Mail op de Python-Dev mailinglist, eerste mail in de discussie gestuurd op "Tue, 1 Oct 2002 21:59:00 +0200".

[6] S. Pedroni. [python-dev] perplexed by mro. `http://mail.python.org/pipermail/python-dev/2002-October/029036.html`, Oct. 2002. Mail op de Python-Dev mailinglist, tweede mail in de discussie gestuurd op "Wed, 2 Oct 2002 00:45:39 +0200".

[7] M. Simionato. The python 2.3 method resolution order. `http://www.python.org/download/releases/2.3/mro/`, 2003. Artikel bezocht in mei 2010.

[8] G. van Rossum. [python-dev] re: my proposals about mros (was: perplexed by mro). `http://mail.python.org/pipermail/python-dev/2002-October/029181.html`, Oct. 2002. Mail op de Python-Dev mailinglist, voorlaatste mail in de discussie gestuurd op "Fri, 04 Oct 2002 15:31:13 -0400".

[9] G. van Rossum. Unifying types and classes in python 2.2. `http://www.python.org/download/releases/2.2/descrintro/`, 2002. Artikel bezocht in mei 2010. Originele versie van dit document.

[10] G. van Rossum. Unifying types and classes in python 2.2. `http://www.python.org/download/releases/2.2.3/descrintro/`, 2003. Artikel bezocht in mei 2010. Nieuwste versie van dit document.

[11] G. van Rossum. Pep 252: Making types look more like classes. `http://www.python.org/dev/peps/pep-0252/`, June 2007. Artikel bezocht in mei 2010.

[12] G. van Rossum. Pep 253: Subtyping built-in types. `http://www.python.org/dev/peps/pep-0253/`, June 2007. Artikel bezocht in mei 2010.