Bachelorscriptie
Informatica

Radboud Universiteit

Een ownership based garbage-collector voor de taal cgt

Auteur:
Charlie Gerhardus
s3050009

Inhoudelijk begeleider:
dr. P.M. Achten
p.achten@cs.ru.nl

[Tweede lezer:]
dr. P. Koopman
pieter@cs.ru.nl

28/8/2015

Samenvatting

Deze scriptie heeft als doel om een nieuwe garbage-collector voor de programmeertaal cgt te definiëren. De huidige versie maakt gebruik van reference-counting. Deze techniek is niet altijd in staat om al het geheugen vrij te geven. Verschillende bestaande oplossingen worden beschouwd om tenslotte een nieuwe garbage collector te definiëren die eigenschappen van een reference-counting en ownership based garbage collector combineert. Deze nieuwe garbage-collector is in staat om cyclische referenties te detecteren en op te ruimen zonder dat de programmeur de controle over het moment waarop garbage wordt opgeruimd verliest. Tot slot gaan we kort in op een mogelijke implementatie van deze nieuwe garbage-collector.

Inhoud

Hoofdstuk 1

Inleiding

Het oorspronkelijke doel van de cgt programmeertaal was om als scripting taal voor een game engine te dienen. Gedurende ontwikkeling is er gekozen om ook een stand-alone virtual machine te bouwen. Vandaag de dag resteert enkel de stand-alone versie van cgt. Omdat in eerste instantie scripting van games een belangrijke factor was in het ontwerp van cgt is er voor gekozen om een objectgeoriënteerde taal te ontwikkelen waarmee gemakkelijk complexere taken gerealiseerd kunnen worden. Dit betekend onder andere dat de runtime envoriment garbage collection afhandelt en het werken met rijen van een dynamische lengte in de taal ingebouwd is.

De huidige virtueele machine gebruikt reference-counting om het geheugen te beheren. Het grote nadeel van reference-counting is dat het niet in staat is om cyclische referenties te detecteren en op te ruimen. Hierdoor zal een programma dat cyclische referenties bevat het desbetreffende geheugen niet vrijgeven, in het ergste geval kan dit een runtime-error opleveren als het systeem geen geheugen meer beschikbaar kan stellen voor het programma. Deze scriptie tracht een garbage-collector te definiëren die controle biedt over wanneer een object vrijgegeven wordt en tevens in staat is om onbereikbare cyclische referenties te detecteren en op te ruimen.

In deze scriptie komen de volgende onderwerpen aan bod.

- Beschrijving cgt programmeertaal.
- Het cgt runtime omgeving geheugenmodel.
- Huidige cgt garbage collector en zijn tekortkomingen.
- Vergelijking van verschillende bestaande garbage-collectors.
- Nieuwe ownership based garbage-collector.

De volgende garbage collectors worden geanalyseerd.

- Reference-counting.
- Mark 'n Sweep.
- Stop-and-copy.
- On-the-fly.
- Ownership based.
- Hybride vormen van verschillende garbage collectors.

Omdat geen van de geanalyseerde garbage collectors voldoet aan de eisen die cgt stelt (zie 3.3) wordt er een nieuwe garbage collector geformuleerd. Deze hybride vorm van een ownership based garbage collector in combinatie met een tracing collector is zowel in staat de programmeur volledige controle te geven over het moment waarop de deconstructor van een classe wordt en om cyclische referenties te lokaliseren en op te ruimen.

Hoofdstuk 2

De taal

Dit hoofdstuk beschrijft het gebruik van de taal cgt. Een volledig overzicht van de grammatica is te vinden in de appendix A.2.

2.1 Notatie

2.1.1 Sleutelwoorden

Cgt is een Nederlandstalige programmeertaal, dit is terug te zien in de volgende tabel die verschillende sleutelwoorden uit de taal naast zijn C++ tegenhanger zet.

cgt	C++
Rubriek	namespace
Bijlage	# include
Sjabloon	class
niets	NULL
als	if
anders	else
zolang	while
alle	for
opnieuw	continue
ontspring	break
verlaat	return
nieuw/nieuwe	new

2.1.2 Commentaar

Er is de mogelijkheid om commentaar aan de code toe te voegen die een regel beslaat of een blok aanduidt. Een commentaarblok kan meerdere regels beslaan.

```
1 # commentaar tot einde van regel
2
3 >#  commentaar
4       dat
5       meerdere
6       regels
7       beslaat. #<
8
9 x : Z >#  of een deel #< .
```

2.1.3 Leestekens

Omdat cgt een Nederlandstalige programmeertaal is verschillen de tekens voor de operatoren ten opzichte van Engelstalige programmeertalen. Alle statements worden niet geëindigd met een in C++ gebruikelijk ';' maar met een '.'. ';' is het scheidingsteken en niet ',', dit komt omdat we komma getallen met een komma en niet met een punt noteren (3,14 in tegenstelling tot 3.14).

2.2 Hello World

```
1  # huidige rubriek instellen
2  Rubriek hello_world.
3
4  # entry point van alle cgt applicaties
5  #  pr is een rij met alle commandline parameters
6  start(pr:[Tekst]) : B {
7      # schrijf naar het scherm
8      ~scherm..schrijf("Hallo Wereld!").
9
10     # termineer zonder error
11     verlaat ja.
12 }
```

2.3 Rubrieken

Alle cgt code is ingedeeld in rubrieken. Deze zijn te vergelijken met namespaces in C++ of packages in Java. Wanneer we een symbool schrijven (een functie, variabele of sjabloon) dan zoeken we deze in de lokale rubriek. Wanneer we naar een symbool willen refereren die zich niet in de lokale rubriek bevindt (en de lokale rubriek is niet de oorsprong) dan moeten we dit aanduiden door de naam van het symbool vooraf te gaan met $\sim$. Een

rubriek kan op zijn beurt weer subrubrieken bevatten. Tussen de naam van een rubriek en de naam van een subrubriek of symbool dat zich in de rubriek bevindt schrijven we '..'. Elk bronbestand kan maar 1 rubriek beschrijven, deze dient aan het begin van het bronbestand te worden opgegeven. Als we dit niet doen dan worden de symbolen toegevoegd aan de oorsprong (∼).

```
1  # we zijn de rubriek ~naam
2  Rubriek naam.
3
4  # we zijn de rubriek ~naam..subnaam
5  Rubriek naam..subnaam .
6
7  ...
8  # roep functie in ~overig aan
9  ~overig..mijn_functie().
10
11 # roep functie in subrubriek van huidige rubriek aan
12 # dit refereerd naar ~naam..subnaam..subrubriek
13 subrubriek..mijn_functie()
14 ...
```

Wanneer we de rubriek opgeven aan het beging van een bronbestand hoeven we deze dus niet met ∼ vooraf te gaan omdat we hier altijd vanuit de oorsprong werken.

2.4 Functies en variabelen

Globaal gedeclareerde variabele en alle functies zijn in alle bron bestanden beschikbaar. Er dus geen mogelijkheid om functionaliteit af te schermen.

```
1  # Naam : Type .
2  x : Z.
3
4  # Naam ( Parameternamen en types ) : Type { Code }
5  f (x : Z) : Z {
6      verlaat x*2.
7  }
```

We kunnen globale variabele een initiële waarde geven. Dit doen we door de declaratie te verlengen met de toekenningsoperator '='. Alle initiële waarden worden berekend voordat de *start*() functie wordt uitgevoerd.

```
1  y : Z = 10.
```

Wanneer globale variabelen van elkaar afhankelijk zijn zal er worden geprobeerd de volgorde waarin deze geïnitialiseerd worden aan te passen. In het geval van circulaire afhankelijkheid geeft de compiler een fout.

```
1  # dit mag
2  a : N = b.
3  b : N = 10.
4  # a en b zijn nu 10
5
```

```
6 # dit mag niet
7 c : N = d.
8 d : N = c.
9 # fout: c en d zijn van elkaar afhankelijk.
```

Voor alle variabelen geldt dat wanneer zij geen initiële waarde meegegeven krijgen het geheugen gevuld wordt met nullen. Voor referenties naar dynamische instanties (rij en sjabloon instanties) betekent dit dat het adres gelijk is aan *niets*.

2.4.1 Constanten

Naast globale variabelen hebben we ook de mogelijkheid om constante waarden te declareren. Dit gaat bijna op dezelfde manier als bij globale variabelen met als verschil dat we niet de toekenningsoperator '=' gebruiken maar de constante toekenningsoperator ':='. Referenties naar constanten worden tijdens het vertalen naar bytecode al opgelost. Dit betekent dat wanneer we gebruik maken van een constante die zich in een geïmporteerde bibliotheek bevindt en we deze in de bibliotheek veranderen deze verandering niet zonder hercompilatie doorgevoerd wordt in andere bibliotheken/programma's die van deze constante gebruik maken.

```
1 # constante
2 PI : R := 3,14.
```

In de toekenning van constanten mogen we alleen gebruik maken van andere constanten. Tevens mogen constanten ook alleen ingebouwde type gebruiken (zie sectie 2.5). Voor constanten geldt net als bij variabele dat de volgorde van initialisatie wordt veranderd wanneer dit nodig is. Tevens geeft een circulaire afhankelijkheid ook een fout.

2.5 Typen

2.5.1 Ingebouwde typen

Deze typen zijn ingebouwd in de virtuele machine en zijn de meest atomaire objecten die we kunnen gebruiken.

Tabel 2.1: Ingebouwde typen

Symbool	Dimensie	Omschrijving
Geen	0 bytes	Dit type kan gebruikt worden om aan te geven dat we geen waarde verlaten.
B	4 bytes	Booleaanse waarde (*ja* of *nee*).
N8	1 bytes	8 bits natuurlijk getal.
N16	2 bytes	16 bits natuurlijk getal.
N32	4 bytes	32 bits natuurlijk getal.
N64	8 bytes	64 bits natuurlijk getal.
N	4 bytes	32 bits natuurlijk getal (synoniem voor N32).
Z8	1 bytes	8 bits geheel getal.
Z16	2 bytes	16 bits geheel getal.
Z32	4 bytes	32 bits geheel getal.
Z64	8 bytes	64 bits geheel getal.
Z	4 bytes	32 bits geheel getal (synoniem voor Z32).
R32	4 bytes	32 bits reëel getal.
R64	8 bytes	64 bits reëel getal.
R	4 bytes	32 bits reëel getal (synoniem voor R32).
Teken	1 bytes	8 bits natuurlijk getal, kan 1 UTF-8 character representeren.
Tekst	8 bytes	Adres van een reeks tekens, op 64bit machines zijn hier 8bytes voor nodig.

Rijen

Rijtypen komen in twee varianten. De eerste vorm wordt gebruikt tijdens het declareren van symbolen en bevat geen informatie over de lengte van een rij. Wanneer we een nieuwe rij alloceren m.b.v. de *nieuwe/nieuw* operator gebruiken we de tweede vorm waarin we ook de lengte aangeven. Alle rijen hebben een ingebouwde variabele $rij \rightarrow lengte$ waarmee we het aantal elementen in de rij kunnen verkrijgen en een ingebouwde operatie

$rij \rightarrow verstel(n)$ die de lengte van de rij verstelt tot n elementen.

```
1  # eerste vorm
2  var1 : [Z ] .
3
4  # tweede vorm, deze mag enkel in combinatie met de nieuwe
       operator
5  # gebruikt worden.
6  var2 : [Z] = nieuwe [10\Z ] ( ) . # () altijd verplicht
7
8  # een rij van natuurlijke getallen
9  rij : [N].
10
11 # een rij van rijen natuurlijke getallen
12 rij2D : [[N]].
```

Teksten

Teksten hebben een apart type en zijn niet hetzelfde als een rij met tekens ($[Teken]$). In het geheugen representeren we teksten als een reeks tekens afgesloten met een 0, dit getal representeert dus ook geen geldig teken. Net als rijen hebben teksten een *lengte* veld. Dit veld bevat het aantal tekens in de tekenreeks zonder de terminerende 0. Wanneer de tekstreferentie gelijk is aan *niets* resulteert dit in een lengte van 0.

Als we de index operator [] op een waarde van het type $Tekst$ toepassen kunnen we 1 teken uit de tekst lezen. Het is niet mogelijk om de tekst m.b.v. deze operator te manipuleren.

Net als getallen kunnen we twee teksten vergelijken m.b.v. de == operator. Het resultaat van deze berekening is een booleaanse waarde die ja is wanneer de teksten gelijk zijn.

Sjablonen

Natuurlijk zijn alle sjablonen eveneens te gebruiken als een type. Dit type duidt een referentie naar een instantie van het desbetreffende sjabloon aan. De daadwerkelijke instantie kunnen we alloceren met behulp van de *nieuw/nieuwe* operator. Voor meer informatie over het definiëren van sjablonen zie sectie 2.7. Meer informatie over het instantiëren van een sjabloon staat in sectie 2.7.7.

Functiereferenties

Functiereferenties zijn variabelen die het adres van een functie kunnen bevatten. We kunnen deze variabelen daarna als gewone functie aanroepen. Het type bevat de verlaat en parameter typen. In een toekenning kunnen alleen functies die hier precies mee overeen komen of andere functie referenties met hetzelfde type gebruikt worden. Het type bestaat uit het verlaat type

gevolgd door $\leftarrow$ en alle parameter typen omvat met (en). In feite worden alle operaties van een sjabloon ook als variabele met een functiereferentie type in de instantie bewaard.

```
1 # een functie
2 f (x :Z; y :Z) : Z {
3     verlaat x*y.
4 }
5
6 # een functie referentie type
7 var : Z <- (Z;Z) = f.
8
9 # var (10;10) , resulteert nu in f (10;10).
```

Typeringssysteem

Onderstaande tabel geeft het type van het resultaat voor berekeningen met de operatoren $\oplus \in \{+, -, *, /, \%\}$. In het geval van $Tekst$ kan de + operator gebruikt worden om twee teksten te concateneren.

Tekst	+	Tekst	$->$	Tekst
R	$\oplus$	x	$->$	R, met $x \in \{B, N, Z, R\}$
x	$\oplus$	R	$->$	R, met $x \in \{B, N, Z, R\}$
Z	$\oplus$	x	$->$	Z, met $x \in \{B, N, Z\}$
x	$\oplus$	Z	$->$	Z, met $x \in \{B, N, Z\}$
N	$\oplus$	x	$->$	N, met $x \in \{B, N\}$
x	$\oplus$	N	$->$	N, met $x \in \{B, N\}$
B	$\oplus$	B	$->$	N

Voor de binaire operatoren $\oplus \in \{\&, |, !|\}$ wordt in onderstaande tabel het resultaat aan de hand van de types van de operanden gegeven.

Z	$\oplus$	Z	$->$	Z
N	$\oplus$	x	$->$	N, met $x \in \{N, Z\}$
x	$\oplus$	N	$->$	N, met $x \in \{N, Z\}$

Voor de vergelijkende operatoren (==, !=, >, >=, <, <=) is het resultaat altijd van het type B. Voor teksten zijn enkel de gelijk en ongelijkheidsvergelijkingen mogelijk. Ook voor de logische operatoren (en, of) geldt dat het resultaat altijd van het type B is.

Meerdere booleaanse waarde kunnen in een conditie gecontroleerd worden. Dit kan met de en en of operator. In het geval van de en operator is het resultaat alleen waar als zowel de vergelijking links als rechts van de operator waar is, voor of geldt dat er enkel één van de vergelijkingen waar

hoeft te zijn. Wanneer een conditie niet langer waar kan zijn wordt de rest niet meer uitgevoerd. Dit kan nuttig wanneer we willen controleren of een sjabloon referentie geldig is en wat het resultaat van een operatie is.

```
1 als (!foo of !foo->opslaan())
2 {
3     ~scherm..schrijf("fout tijdens opslaan").
4 }
```

Wanneer *foo* in het bovenstaande voorbeeld gelijk is aan *niets* zal de operatie *opslaan*() niet aangeroepen worden.

Waarden van het type B, N, Z of R kunnen naar elk van deze typen geconverteerd worden. Voor sjablonen geldt dat we een subklasse altijd naar zijn superklasse kunnen converteren. Dit gebeurt bijvoorbeeld tijdens een toekenning of het doorgeven van functie parameters.

Converteren

We kunnen m.b.v. enkele functies uit de standaard bibliotheek getallen als tekst representeren en vice versa. De conversie functies zijn te vinden in de rubriek van het doel type én van het bron type.

```
1 i_nummer : N = ~N..van_Tekst("10").
2 t_nummer : Tekst = ~Tekst..van_N(i_nummer).
3
4 i_nummer : N = ~Tekst..als_N("10").
5 t_nummer : Tekst = ~N..als_Tekst(i_nummer).
```

2.6 $start(pr : [Tekst]) : B$

De *start*() functie is het beginpunt van een cgt programma. Nadat alle globale variabelen zijn geïnitialiseerd wordt deze functie aangeroepen. De waarde waarmee deze functie wordt verlaten wordt gebruikt om het besturingssysteem mede te delen of het programma succesvol is getermineerd. In tegenstelling tot de standaard *main*() zoals we die in C/C++ kennen duiden wij een succes aan met $ja/1$ en betekent $nee/0$ dat het programma niet succesvol was. De parameter *pr* is een rij met teksten die alle parameters bevat die aan het programma worden meegeven.

2.7 Sjablonen

Sjablonen zijn verzamelingen van variabelen en operaties waar een instantie van gemaakt kan worden m.b.v. de *nieuw* of *nieuwe* operator. De compiler genereert voor beide operatoren dezelfde code en het verschil is puur cosmetisch. Wanneer we een instantie aanmaken wordt de *init*() operatie van het sjabloon aangeroepen. Voordat we de plek in het geheugen waar

de sjabloon instantie zich bevindt vrijgeven wordt eerst de *deinit*() operatie aangeroepen.

2.7.1 Member variabele

Variabelen die onderdeel uitmaken van een sjabloon definiëren we op dezelfde manier als globale variabele met als uitzondering dat we geen initiële waarde kunnen toekennen. Het initialiseren van de variabele gebeurt in de *init*() operatie (zie secties 2.7.3, 2.7.4 en 2.7.5).

2.7.2 Operaties

Operaties definiëren we hetzelfde als functies. Het verschil zit hem erin dat alle operaties een extra parameter hebben genaamd *ik*. Deze bevat een referentie naar de instantie waarvoor we de operatie aanroepen. We hoeven deze parameter nooit zelf op te geven want de compiler zorgt ervoor dat deze automatisch wordt toegevoegd.

2.7.3 $init() : B$

init is een optionele operatie die wordt aangeroepen wanneer een nieuwe instantie aangemaakt wordt. De verlaat waarde (een boolean) bepaalt de waarde van de *geldig* member variabele. Wanneer we deze operatie parameters geven is het vereist om deze bij het aanroepen van de *nieuw*/*nieuwe* operator mee te geven.

2.7.4 $deinit() : Geen$

deinit is een optionele operatie die aangeroepen wordt voordat een sjabloon vernietigd wordt. De code in deze operatie mag er vanuit gaan dat alle instanties die eventueel members zijn van deze instantie pas na het voltooien van deze operatie worden opgeruimd.

2.7.5 Geldigheid

Elke sjabloon instantie heeft een member variabele *geldig* van het type B. Het resultaat van de *init*() operatie wordt hierin bewaard en kan worden gebruikt om aan te geven dat het initialiseren van een sjabloon instantie is gefaald. De geldigheid van een instantie kan enkel gelezen en nooit handmatig veranderd worden. Sjablonen die geen *init*() operatie hebben zijn altijd geldig. Wanneer een sjabloon als ongeldig wordt geïnitialiseerd kan deze nog steeds worden gebruikt. Het is dus mogelijk om bijvoorbeeld meer informatie over een fout beschikbaar te stellen.

```
1 # sjabloon dat altijd ongeldig is
2 Sjabloon Foo
```

```
3  {
4      fout : Tekst.
5
6      init() : B {
7          ik->fout = "Foo->init() gaf een fout".
8          verlaat nee.
9      }
10 }
11
12 # start functie die geldigheid controleert
13 start ( pr : [ Tekst ] ) : B {
14     foo : Foo = nieuwe Foo ( ) .
15     als (foo->geldig != ja)
16     {
17         ~scherm..schrijf(foo->fout).
18         verlaat nee .
19     }
20
21     #foo is geldig
22     verlaat ja .
23 }
```

2.7.6 Overerven

We kunnen een nieuw sjabloon definiëren die de inhoud van een ander sjabloon overerft. Dit doen we door na de naam van het sjabloon de $\leftarrow$ operator te schrijven gevolgd door de naam van de superklasse omvat met (en). Multiple-inheritance is niet toegestaan. Wanneer we vanuit een operatie een operatie van de superklasse willen aanroepen gebruiken we $Vader$ i.p.v. ik. Zo is het dus mogelijk om van operaties die we overschrijven toch de implementatie in de superklasse te bereiken. Met $Vader \rightarrow init(...)$ is het dus mogelijk om de init operatie van de superklasse aan te roepen, dit gebeurt *niet* automatisch!

```
1  Sjabloon Foo
2  {
3      x : Z.
4      init() : B {
5          ik->x = 10.
6          verlaat ja .
7      }
8
9      toon() : Geen {
10         ~scherm..schrijf(" x : "+~Tekst..van_Z(ik->x)).
11     }
12 }
13
14 Sjabloon Bar <-(Foo)
15 {
16     y : Z.
17     init() : B {
```

```
18          Vader->init().
19
20          ik->y = 100.
21          verlaat ja .
22      }
23
24      toon() : Geen {
25          Vader->toon().
26          ~scherm..schrijf(" y : "+~Tekst..van_Z(ik->y)).
27      }
28  }
29
30  # Bar->toon() toont het volgende op het scherm
31  # x : 10
32  # y : 100
```

Wanneer we een operatie niet overschrijven zal automatisch de implementatie uit de superklasse overgenomen worden.

2.7.7 Instantiëren

Instantiëren doen we met behulp van de *nieuw/nieuwe* operator. Deze wordt gevolgd door de naam van het sjabloon en de parameters voor de *init*() operatie, indien het sjabloon deze heeft.

```
1  # voor een sjabloon zonder init() parameters
2  x : Foo = nieuwe Foo().
3
4  # voor een sjabloon met init() parameters
5  y : Bar = nieuwe Bar(x).
```

2.8 Rijen instantiëren

Met behulp van de *nieuw/nieuwe* operator kunnen we rijen instantiëren. We moeten hier nu wel de lengte van de rij aan het type toevoegen. Het is niet toegestaan om rijen met meer dan 1 dimensie te alloceren. Om deze rijen toch te instantiëren dienen we eerst een één dimensionale rij te alloceren en dit dan te herhalen voor elke index in de rij.

$rij->lengte$

Leest de rijlengte. Voor rij referenties die gelijk zijn aan *niets* resulteert dit in een lengte van 0.

$rij->verstel(n)$

Deze ingebouwde operatie zal de lengte van een rij veranderen. Voor een rij referentie die gelijk is aan *niets* resulteert dit in een runtime-error. Het is daarom belangrijk om te controleren of dit niet het geval is.

```
1  rij : [N] = nieuwe [2\N](). # () niet optioneel!
2
3  # rij is nu een rij met 2 elementen
4  rij[0] = 10.
5  rij[1] = 20.
6
7  # controleer of rij niet niets is
8  als (rij != niets)
9  {
10     # maak de rij 1 element langer
11     rij->verstel(3).
12
13     # stel dit nieuwe element in
14     rij[2] = 30.
15 }
```

Als we een rij groter maken dan worden alle nieuwe elementen $0/niets$ gemaakt. Als we een rij verkleinen geven we het niet gebruikte geheugen vrij en indien we te maken hebben met rij of sjabloon referenties verlagen we de reference-count van alle elementen die uit de rij worden verwijderd. Als de rij elementen van het type $Tekst$ bevat en er worden elementen aan het einde van de rij verwijderd dan wordt ook hiervan het geheugen vrijgegeven.

2.9 Controle structuren

2.9.1 als, anders, anders als

als (c) { s }

Wanneer de conditie/berekening c in ja resulteert worden de statements s uitgevoerd.

als (c_1) { s_1 }
anders { s_2 }

Wanneer de conditie/berekening c in ja resulteert voeren we de statements $s1$ uit, als c in *nee* resulteert worden de statements $s2$ uitgevoerd. Na *anders* mag er ook een nieuwe *als* (c) { s } geschreven worden welke wordt uitgevoerd als de conditie geldt en de oorspronkelijke conditie faalt.

2.9.2 zolang

zolang (c) { s }

De statements s worden herhaaldelijk uitgevoerd zolang de conditie c in ja resulteert.

2.9.3 alle

alle (init ; c ; stap) { s }

Allereerst wordt *init* uitgevoerd, dit *moet* een toekenning van een variabele zijn. Daarna wordt de conditie *c* berekend, als deze *ja* is worden de statements *s* uitgevoerd en daarna de toekenning *stap*. Na het uitvoeren van *stap* wordt de conditie *c* opnieuw bekeken en zolang deze in *ja* resulteert *s* en *stap* uitgevoerd.

2.9.4 ontspring

Wanneer we ons in een *zolang* of *alle* lus bevinden kunnen we *ontspring* gebruiken om de lus te verlaten.

2.9.5 opnieuw

In een *zolang* lus zullen we de conditie opnieuw bekijken en de lus dan eventueel vervolgen. Wanneer we ons in een *alle* lus bevinden zal eerst de *stap* uitgevoerd worden alvorens we de conditie opnieuw evalueren.

2.9.6 verlaat

Verlaat komt in twee vormen en kan gebruikt worden om een functie body te verlaten. Wanneer we te maken hebben met een functie die *Geen* waarde verlaat gebruiken we de kortste variant.

verlaat.

Als we te maken hebben met een functie die wel een waarde verlaat gebruiken we.

verlaat b .

Hier is *b* een berekening die in een waarde van het juiste type resulteert dat als resultaat aan de code die ons aanroept wordt doorgeven.

2.10 Geheugen model

De virtuele machine brengt al het geheugen onder in de volgende secties.

ROM Hier bevinden zich alle instructie opcodes en constanten waarde. Alle functies en operaties verwijzen naar een positie in het ROM geheugen gebied waar de gegenereerde instructies zich bevinden.

RAM Gebied met vaste lengte. Alle variabelen die binnen een rubriek worden gedeclareerd hebben een adres dat ligt in het RAM gebied.

Stapel Tijdelijke waarde en parameters/resultaat instructies bewaren we hier. Lokale variabelen worden eveneens op de stapel bewaard.

Heap Alle dynamisch gealloceerde objecten (rijen en sjabloon instanties) worden in dit gebied geplaatst. Deze instanties worden m.b.v. reference-counting beheerd. Deze vorm van garbage-collection wordt beschreven in 4.1.1. Telkens wanneer we het adres van een sjabloon of rij instantie op de stapel leggen verhogen we de reference-count (zie tabel 5.2 en 2.5). Wanneer het adres van de stapel gepakt wordt zonder deze naar een andere plaats in het geheugen te verplaatsen, verlagen we de reference-count. In het geval dat het adres wel op een nieuwe positie bewaard wordt (in het geval van een assignment) blijft de reference-count onveranderd. Indien de reference-count 0 wordt, wordt de instantie vrijgegeven.

2.10.1 Rijen en sjablonen in het geheugen

Wanneer we instanties van rijen en sjablonen maken bevatten deze informatie die we gebruiken om de instanties op het juiste moment vrij te geven. Elke referentie met een rij of sjabloon type verwijst naar een instantie met eerst een 4byte lange reference-count, 1 identificatie byte gevolgd door de daadwerkelijke inhoud van de instantie. In het geval van rijen is dit een reference-count, identificatie byte, de rij lengte en een adres naar de elementen waar de rij uit bestaat. Sjabloonreferenties bevatten naast de reference-count en identificatie byte het relatieve adres van het element dat de init functie bevat (0h0000 betekent geen init functie), het relatieve adres van de deinit functie (kan ook leeg zijn), gevolgd door alle members (variabelen en operaties).

Tabel 2.4: Instantie Typen

Waarde	Type
0h0F	Sjabloon instantie.
0h80	Rij van interne typen (N8,N16...).
0h88	Rij van teksten.
0h8F	Rij van sjabloon of rij instanties.

Het is belangrijk om onderscheid te maken tussen rijen van basis typen, teksten en sjablonen of rijen. Wanneer de reference-count van een rij 0 bereikt zal in het geval van een rij met sjablonen of rijen de reference-count van alle elementen in de rij met 1 verlaagd moeten worden. In het geval van

teksten is het ook nodig om alle elementen vrij te geven omdat teksten niet meer zijn dan een adres naar een met 0 getermineerde reeks tekens.

Tabel 2.5: Rij Instantie Codering

Adres	Lengte	Inhoud
0h0000	4	Reference-count.
0h0004	1	Id (zie Instantie Typen).
0h0005	4	Aantal elementen in rij.
0h0009	8	Adres van rij inhoud.

Het laatste element bevat het adres van de daadwerkelijke rij in het geheugen.

Tabel 2.6: Sjabloon Instantie Codering

Adres	Lengte	Inhoud
0h0000	4	Reference-count.
0h0004	1	Id (zie Instantie Typen).
0h0005	4	Positie van init operatie.
0h0009	4	Positie van deinit operatie.
0h000D	8	Adres van gegenereerde deref operatie.
0h0015	4	'geldig' element van dit sjabloon.
0h0019	n	Sjabloon elementen.

Voor elk sjabloon bevat het bytecode programma een voorbeeldinstantie. Nadat we een programma in het geheugen hebben geladen bevat deze voorbeeldinstantie de adressen van alle operaties en zijn alle variabelen met 0 geïnitialiseerd. Wanneer we met behulp van *nieuw*/*nieuwe* een sjabloon instantiëren kopiëren we dit voorbeeld naar een locatie in het geheugen en in het geval dat er een *init*() operatie beschikbaar is roepen we deze aan.

```
1  Sjabloon Foo
2  {
3      x : N.
4
5      init() : B {
6          ik->x = 0.
7          verlaat ja.
8      }
9
10     toon() : Geen {
11         ~scherm..schrijf( ~N..als_tekst(ik->x) ).
12     }
```

```
13
14      verhoog() : Geen {
15          ik->x += 1.
16      }
17  }
```

Bovenstaande instantie wordt dus als volgt in het geheugen bewaard.

Tabel 2.7: Foo instantie codering

Adres	Lengte	Waarde
0h0000	4	1
0h0004	1	0h0F
0h0005	4	0h0000001D
0h0009	4	0h00000000
0h000D	8	-
0h0015	4	0
0h0019	4	0 (variable x)
0h001D	8	adres $Foo->init()$
0h0025	8	adres $Foo->toon()$
0h002D	8	adres $Foo->verhoog()$

Wanneer we een overerving van Foo maken nemen we de voorbeeld-instantie van dit sjabloon en voegen nieuwe variabelen en operatoren aan het einde toe. Wanneer we een operatie overerven dan overschrijven we het element dat het adres van de operatie bevat i.p.v. een element aan het voorbeeld toe te voegen. Dit geld niet voor de $init()$ operatie omdat deze qua parameters niet altijd met de oorspronkelijke definitie overeenkomt.

```
1  Sjabloon Bar <-(Foo)
2  {
3      init() : B {
4      }
5
6      verhoog() : Geen {
7          ik->x += 2.
8      }
9  }
```

Het bovenstaande voorbeeld resulteert dus in de volgende instantie.

Tabel 2.8: Foo instantie codering

Adres	Lengte	Waarde
0h0000	4	1
0h0004	1	0h0F
0h0005	4	0h00000035
0h0009	4	0h00000000
0h000D	8	-
0h0015	4	0
0h0019	4	0 (variable x)
0h001D	8	adres $Foo->init()$
0h0025	8	adres $Foo->toon()$
0h002D	8	adres $Bar->verhoog()$
0h0035	8	adres $Bar->init()$

Door instanties op deze manier te coderen kan een sjablooninstantie probleemloos gecast worden naar zijn supertype omdat de velden die overeenkomen automatisch naar de juiste operaties verwijzen.

Hoofdstuk 3

Probleemstelling

3.1 Huidige garbage-collector

Op dit moment maakt de taal gebruik van een reference-counting algoritme om te bepalen welke instanties vrijgegeven kunnen worden. Een beschrijving van dit algoritme is te vinden in sectie 4.1.1. Hierdoor is het moment waarop een instantie garbage wordt ook het moment waarop de deconstructor wordt aangeroepen. Een voorbeeld van een situatie waarin het van belang is dat de deconstructor op het juiste moment wordt aangeroepen is hieronder gegeven.

```
1  g_teller : N = 0.
2
3  Sjabloon Verhoog
4  {
5      nummer : N.
6
7      init() : B {
8          ik->nummer = g_teller.
9          verlaat ja.
10     }
11
12     deinit() : Geen {
13         g_teller = ik->nummer.
14     }
15
16     stap() : Geen {
17         ik->nummer += 1.
18     }
19 }
20
21 start(pr:[Tekst]) : B {
22     foo : Verhoog = nieuwe Verhoog().
23
24     foo->stap().
25     ...
26     foo->stap().
27
```

```
28      foo = niets.
29      foo = nieuwe Verhoog().
30
31      foo->stap().
32
33      verlaat ja.
34  }
```

Op regel 28 zal de reference-count van de eerste instantie van $Verhoog$ 0 worden. Hierdoor zal $Verhoog- > deinit()$ worden aangeroepen en zal de waarde van g_teller veranderen. Hierna zal de nieuwe instantie van $Verhoog$ beginnen met de laatste waarde van de eerste instantie die op dat moment in g_teller te vinden is. In programmeertalen die gebruik maken van periodiek aangeroepen garbage-collectors (zoals Java) zal het bovenstaande voorbeeld niet altijd het gewenste resultaat geven. Wanneer de garbage-collector niet is uitgevoerd tussen regel 28 en 29 zal de nieuwe instantie van $Verhoog$ met de beginwaarde van g_teller (0) geïnitialiseerd worden i.p.v. de waarde waar de eerste instantie van $Verhoog$ mee eindigt. Omdat het wel mogelijk is dat de garbage collector tussen regel 28 en 29 uitgevoerd wordt kan men niet met zekerheid zeggen wat de waarde van g_teller is op het moment dat de nieuwe instantie wordt geïnitialiseerd.

3.2 Het probleem

Een probleem met reference-counting is dat cyclische referenties resulteren in instanties die nooit vrijgegeven worden. Traditionele oplossingen voor dit probleem zorgen ervoor dat de programmeur geen controle meer heeft over wanneer een deconstructor wordt aangeroepen, iets wat in cgt niet gewenst is.

```
1  # voorbeeld cyclische referentie
2
3  Sjabloon Foo
4  {
5      ref : Foo.
6  }
7
8  start(pr:[Tekst]) : B {
9      # maak een instantie van Foo
10     foo : Foo = nieuwe Foo().
11
12     # zorg dat foo een referentie naar zichzelf bevat
13     foo->ref = foo.
14
15     verlaat ja.
16 }
```

Na het verlaten van $start()$ zal de reference-count van de instantie foo 1 zijn. Dit komt omdat nadat $verlaat$ in $start()$ de reference-count van foo

verlaagd heeft de member variabele $foo- > ref$ nog bestaat. Om dit te verduidelijken geven we hieronder de codering van de instantie waar foo na regel 10 naar verwijst.

Tabel 3.1: Foo instantie codering na regel 10

Adres	Lengte	Waarde
0h0000	4	1
0h0004	1	0h0F
0h0005	4	0h00000000
0h0009	4	0h00000000
0h000D	8	-
0h0015	4	1
0h0019	8	niets (member ref)

Na regel 13 zal de instantie als volgt in het geheugen bestaan.

Tabel 3.2: Foo instantie codering na regel 14

Adres	Lengte	Waarde
0h0000	4	2
0h0004	1	0h0F
0h0005	4	0h00000000
0h0009	4	0h00000000
0h000D	8	-
0h0015	4	1
0h0019	8	huidige adres - 0h15

De member ref bevat nu een verwijzing naar de instantie zelf en de reference-count is 2 omdat zowel foo als $foo- > ref$ naar de instantie verwijzen. Bij het verlaten van de functie start zal foo out-of-scope raken en wordt de reference-count verlaagd. Omdat de instantie nog een referentie naar zichzelf bevat wordt deze niet 0 maar blijft deze 1. Omdat er geen vanuit de code bereikbare referentie naar de instantie bestaat zal deze nooit worden vrijgegeven.

3.3 Eisen garbage-collector

In deze scriptie zijn we op zoek naar een garbage-collector die aan de volgende eisen voldoet:

- De syntaxis van de taal zo min mogelijk beïnvloeden.
- De semantiek van de taal mag veranderd worden mits er aan de volgende eisen wordt voldaan:
  - Het moet altijd duidelijk zijn wanneer een deconstructor aangeroepen wordt. Dit houdt in dat wanneer een instantie tot garbage wordt gemaakt (moment afhankelijk van het algoritme) de deconstructor onmiddellijk aangeroepen wordt.
- Moet voor zowel klasse instanties als rij instanties werken.
- Moet al het ongebruikte geheugen vrijgeven.

Het is van belang dat de garbage collector zowel sjabloon instanties als rij instanties kan beheren. Dit komt omdat een rij ook onderdeel van een cykel kan zijn zoals in het onderstaande voorbeeld getoond word.

```
1 Sjabloon RijContainer
2 {
3     rij : [RijContainer].
4 }
5
6 rij : [RijContainer] = nieuwe [1\RijContainer]().
7
8 rij[0] = nieuwe RijContainer().
9 rij[0]->rij = rij[0].
```

Tekst waarde blijven gewoon gebruik maken van reference-counting omdat deze geen onderdeel van een cykel uit kunnen maken, omdat deze enkel tekens kunnen bevatten en zo nooit naar een andere instantie kunnen refereren.

Hoofdstuk 4

Discussie oplossingen

In dit hoofdstuk bespreken we verschillende garbage-collectors verduidelijkt met pseudocode. De pseudocode lijkt op een versimpelde vorm van C. De voorbeelden die gebruikt worden om het effect van een garbage-collector op de mutator (het programma) worden in cgt gegeven (zie hoofdstuk 2).

4.1 Bestaande oplossingen

4.1.1 Reference-counting

Reference-counting is al een zeer oude methode om het geheugen mee te beheren. Een van de eerste beschrijvingen wordt gegeven door George A. Collins [2]. Alle instanties hebben een teller (de reference-count). Wanneer een nieuwe instantie geïnitialiseerd wordt stellen we deze teller op 1. Telkens als er een nieuwe referentie naar een instantie wordt gemaakt verhogen we de teller. Wanneer een referentie out-of-scope geraakt verlagen we de teller. Indien de teller op 0 komt betekent dit dat er geen referenties naar deze instantie meer bestaan en we het geheugen kunnen vrijgeven (nadat we de deconstructor hebben aangeroepen). Een efficiëntere vorm van dit algoritme houdt rekening met het feit dat het niet nodig is om altijd de reference count te verhogen/verlagen. Bijvoorbeeld wanneer wij een lokale variabele hebben die refereert naar een instantie en deze als parameter meegeven aan een functie aanroep dan is het niet nodig voor en na de aanroep de reference count te verhogen en verlagen omdat deze altijd groter dan 0 blijft door de lokale referentie die tijdens de aanroep niet kan veranderen. Deze efficiëntere vorm van reference-counting in combinatie met een tracing collector wordt gegeven door Deutsch en Bobrow [6]. Wanneer een programma meerdere threads kan bevatten is het nodig om elke keer als we de reference-count van een instantie veranderen deze te locken. Dit kan onnodig kostbaar worden daarom is het ook mogelijk dit te voorkomen zoals beschreven in [1].

Een voorbeeld van het reference-counting algoritme dat geen rekening houd met thread-safety kan er in pseudocode als volgt uit zien.

```
1  void* new(Type) {
2      ptr = malloc(sizeof(int)+sizeof(Type))
3      ptr->ref_count = 1
4      ((Type)ptr)->init()
5      return ptr
6  }
7
8  void* reference(ptr) {
9      ptr->ref_count += 1
10     return ptr
11 }
```

Wanneer een variable out-of-scope geraakt of wanneer we een return value negeren voeren we een dereference operatie uit, deze ziet er als volgt uit.

```
1  void dereference(ptr) {
2      ptr->ref_count -= 1
3      if (ptr->ref_count == 0)
4          delete ptr
5  }
```

Wanneer men een referentie verandert is het belangrijk niet meteen de reference-count van de oude waarde te verlagen omdat deze in de berekening van de nieuwe waarde gebruikt zou kunnen worden, zoals in het volgende voorbeeld te zien is.

```
1  f = nieuwe Foo().
2
3  f = f.
```

Zouden we hier de reference-count van f verlagen alvorens we de nieuwe waarde berekenen, dan zou de reference-count van f de 0 bereiken en zou de instantie opgeruimd worden.

```
1  f = new(Foo).
2
3  dereference(f).   # reference-count f = 0
4  f = reference(f). # f is hier al opgeruimd
```

Om dit probleem op te lossen introduceren we de volgende operatie voor assignments.

```
1  void assign(src, value) {
2      value = reference(value)
3      dereference(src)
4      src = value
5  }
```

Doordat we nu de reference-count van de nieuwe waarde verhogen alvorens de oude te verlagen voorkomen we dat deze vroegtijdig wordt opgeruimd. Een soortgelijk probleem doet zich voor in het geval dat we enkel een lokale referentie naar een instantie hebben en deze als return-value van een functie/methode gebruiken. Wanneer we de reference-count van alle lokale vari-

abelen zouden verlagen alvorens de return-value in reference-count te verhogen zou deze ook onterecht opgeruimd worden.

Voordelen:

- Garbage wordt opgeruimt op het precieze moment dat het niet langer bereikbaar is.
- De performance-overhead is zo groot als het aantal referenties dat gekopieerd wordt en is niet afhankelijk van het totale aantal instanties in het geheugen.

Nadelen:

- Cyclische referenties worden niet opgeruimd (zie sectie 3.2).

4.1.2 Reference-counting met weak references

Een taal die gebruik maakt van een reference-counting garbage-collector kan het probleem van cyclische referenties verkleinen door een speciaal type te introduceren voor klasse/array referenties. Deze zogenoemde *weak-references* zijn referenties die niet meetellen in de reference count van een instantie. Wanneer men een programma heeft dat een memory leak bevat doordat er een cyclische referentie ontstaat kan in het programma een van de betrokken referenties een weak reference maken zodat het geheugen toch vrijgeven kan worden. Weak references toegepast op Ada wordt beschreven in [3]. Het onderstaande programma levert een memory leak op als we gebruik maken van standaard reference counting.

```
1  Sjabloon Foo
2  {
3      bar : Bar.
4  }
5
6  Sjabloon Bar
7  {
8      foo : Foo.
9  }
10
11 start(pr:[Tekst]) : B {
12     foo : Foo = nieuwe Foo().
13     bar : Bar = nieuwe Bar().
14
15     foo->bar = bar.
16     bar->foo = foo.
17
18     verlaat ja.
19 }
```

Omdat we de relatie tussen *Foo* en *Bar* in het bovenstaande voorbeeld als een parent/child verbinding kunnen zien zouden we een van de twee

referenties als *weak-reference* kunnen beschouwen. Stel dat we de @ operator gebruiken om *weak-references* aan te duiden dan zou het onderstaande voorbeeld niet langer in een memory leak resulteren.

```
1  Sjabloon Foo
2  {
3      bar : Bar.
4  }
5
6  Sjabloon Bar
7  {
8      foo : @Foo.
9  }
10
11 start(pr:[Tekst]) : B {
12     foo : Foo = nieuwe Foo().
13     bar : Bar = nieuwe Bar().
14
15     foo->bar = bar.
16     bar->foo = foo.
17
18     verlaat ja.
19 }
```

Omdat $Bar->foo$ niet meetelt in de reference count van de instantie van Foo zal deze toch de 0 bereiken na het verlaten van de *start*() functie. Het gebruik van *weak-references* betekent dat de programmeur verantwoordelijk is voor het voorkomen van memory leaks en dit niet altijd vanzelf opgelost wordt. Daarnaast is een groot nadeel dat wanneer een instantie wordt opgeruimd er nog *weak-references* in het geheugen te vinden zijn deze nu naar een ongeldige instantie verwijzen, en wanneer deze instanties benaderd worden zal dit in een runtime-error resulteren. Daarnaast is niet altijd mogelijk om cyclische referentie uit een programma te elimineren door enkel wat *weak-references* toe te voegen. Neem bijvoorbeeld het volgende programma.

```
1  Sjabloon Foo
2  {
3      bar : Foo.
4  }
5
6  link(a : Foo; b : Foo) : Geen {
7      c : Foo = nieuwe Foo().
8
9      a->bar = a.
10     b->bar = c.
11 }
12
13 start(pr:[Tekst]) : B {
14     a : Foo = nieuwe Foo().
15     b : Foo = nieuwe Foo().
16
```

```
17      link(a; b).
18
19      verlaat ja.
20  }
```

Dit programma bevat een cyclische referentie, doordat $a->bar$ gelijk is aan a. Als we dit probleem op willen lossen door enkel *weak-references* te introduceren creëren we een nieuw probleem. We gebruiken hier wederom @ om een *weak-reference* type te definiëren.

```
1   Sjabloon Foo
2   {
3       bar : @Foo. >#  weak reference  #<
4   }
5
6   link(a : Foo; b : Foo) : Geen {
7       c : Foo = nieuwe Foo().
8
9       a->bar = a.
10      b->bar = c.
11  }
12
13  start(pr:[Tekst]) : B {
14      a : Foo = nieuwe Foo().
15      b : Foo = nieuwe Foo().
16
17      link(a; b).
18
19      verlaat ja.
20  }
```

Hoewel dit programma geen problematische cyclische referentie meer bevat omdat $a->bar$ niet langer meetelt in de reference-count van de instantie waarnaar a verwijst. Zal bij het verlaten van de functie *start*() de reference-count van deze instantie de 0 bereiken en opgeruimd worden. Bij het verlaten van de functie *link*() wordt de lokale variabele c opgeruimd en zal de reference-count van de instantie waarnaar deze verwijst op 0 terecht komen, omdat de referentie $b->bar$ een weak reference is en niet meetelt in de reference count. De referentie $b->bar$ is nu dus ongeldig. Om het bovenstaande programma toch te laten werken dient er meer veranderd te worden, een herbruikbare oplossing voor het sjabloon *Foo* zou er als volgt uit kunnen zien.

```
1   Sjabloon FooRef
2   {
3       ref_strong : Foo.
4       ref_weak   : @Foo.
5
6       init(ref : Foo; weak : B) : B {
7           als (weak)
8           {
9               ik->ref_weak = ref.
```

```
10          }
11          anders
12          {
13              ik->ref_strong = ref.
14          }
15
16          verlaat ja.
17      }
18
19      reference() : Foo {
20          als (ik->ref_strong)
21          {
22              verlaat ik->ref_strong.
23          }
24
25          verlaat ik->ref_weak.
26      }
27  }
28
29  Sjabloon Foo
30  {
31      bar : RefFoo.
32  }
33
34  link(a : FooRef; b : FooRef) : Geen {
35      c : Foo = nieuwe Foo().
36
37      a->bar = nieuwe RefFoo(a; Ja).
38      b->bar = nieuwe RefFoo(c; Nee).
39  }
40
41  start(pr:[Tekst]) : B {
42      a : Foo = nieuwe Foo().
43      b : Foo = nieuwe Foo().
44
45      link(a; b).
46
47      verlaat ja.
48  }
```

Nu bevat $a->bar$ een referentie naar een instantie van $FooRef$ die een weak-reference naar a bevat. Deze cyclische referentie wordt nu dus nog steeds correct opgeruimd. Daarnaast bevat $b->bar$ nu een referentie naar een instantie van $FooRef$ die een reference-counted referentie naar c bevat. Bij het verlaten van de functie $link()$ zal de reference-count van deze instantie dus niet 0 bereiken en wel bewaard blijven.

Voordelen:

- Maakt het mogelijk cyclische datastructuren te definiëren in een reference-counted omgeving.

- Weak-references vereisen minder garbage-collection tijd dan niet weak-references, wat mogelijk een performance winst kan geven.

Nadelen:

- Programmeur dient handmatig cyclische referenties op te sporen en m.b.v. weak-pointers te elimineren.
- Resulteert in een runtime-error wanneer een weak-reference naar één instantie verwijst waar geen reference-counted referentie meer naar bestaat.

4.1.3 Mark and Sweep

Dit algoritme voert periodiek een stop-the-world uit waarin alle threads stilgezet worden en de garbage collector op zoek gaat naar alle instanties die niet langer bereikbaar zijn voor het programma [12, 10]. Het Mark and Sweep algoritme is één van de eerste tracing garbage collectors en vereist een stop-the-world waarin alle threads van een proces stilgelegd worden voordat de garbage collector zijn werk kan doen. Dit gebeurt in enkele stappen. Eerst markeren we alle instanties als onbereikbaar en gaan daarna vanuit de root (globale en lokale variabelen) op zoek naar alle bereikbare instanties en markeren deze als bereikbaar. Dit wordt gevolgd door een *sweep* waarin we nogmaals alle instanties die zich in het geheugen bevinden doorlopen en alle als onbereikbaar aangemerkte instanties vrijgeven.

```
1  void clear() {
2      foreach instance in Heap do
3          instance->is_reachable = false
4  }
5
6  void mark(node) {
7      foreach ref in node do
8          ref->is_reachable = true
9          mark(ref)
10 }
11
12 void sweep() {
13     foreach instance in Heap do
14         if (!instance->is_reachable)
15             free(instance)
16 }
17
18 void invoke_GC() {
19     pause_all_threads()
20
21     clear()
22     mark(Root)
23     sweep()
24
25     reseume_all_threads()
```

```
26 }
```

Bovenstaande definitie van *mark*() zal nodes waar meerdere referenties naar bestaan opnieuw markeren. Omdat dit niet nodig is kan de *mark*() functie als volgt geoptimaliseerd worden.

```
1  void mark(node) {
2      foreach ref in node do
3          if (ref->is_reachable)
4              continue
5          ref->is_reachable = true
6          mark(ref)
7  }
```

Omdat dit algoritme eerst alle instanties langs gaat is de performance-overhead van de garbage-collector sterk afhankelijk van het aantal instanties in het geheugen. Wanneer men een programma heeft met zeer veel bereikbare instanties terwijl deze nauwelijks gebruikt worden zullen deze toch een significant effect hebben op de duur van een garbage-collection stap.
Voordelen:

- Is in staat om cyclische referenties op te ruimen.

Nadelen:

- Performance afhankelijk van aantal instanties in het geheugen.
- Stop-the-world betekend dat het hele programma periodiek wordt stilgelegd, dit kan een negatieve invloed op de executiesnelheid van een programma hebben.
- Het is niet langer bekend wanneer een instantie wordt opgeruimd, dit gebeurt de eerstvolgende keer dat de garbage collector een stop-the-world forceert nadat een instantie niet langer bereikbaar is.

4.1.4 Reference-counting met periodieke tracing collector

Om reference-counting toch in staat te stellen om cyclische referenties op te ruimen kunnen we deze combineren met een tracing collector. De tracing collector zal periodiek worden uitgevoerd en op zoek gaan naar cyclische referenties die niet meer te bereiken zijn voor het programma en dus opgeruimd dienen te worden. De tracing collector kan een volledig garbage collector algoritme zijn (bijvoorbeeld Mark and Sweep), maar omdat we hier enkel geïnteresseerd zijn in cyclische referenties die niet m.b.v. reference-counting kunnen worden vrijgeven is het ook mogelijk om een algoritme te gebruiken dat enkel cyclische referenties vindt. Local Mark and Sweep is een voorbeeld van een algoritme dat dit kan zoals aangegeven in [4]. Een voorbeeld van reference-counting in combinatie met een periodieke tracing collector wordt beschreven in [7]. Het is ook mogelijk om een zeer beperkt

aantal bits te gebruiken voor de reference-count. Wanneer het verhogen van de reference-count in een overflow van het reference count veld resulteert zal de desbetreffende instantie alleen nog maar met behulp van de tracing garbage collector opgeruimd worden. De gedachte hierachter is dat de meeste instanties maar een zeer beperkt aantal referenties heeft en snel uit het geheugen zal verdwijnen, terwijl instanties met veel referenties in de meeste gevallen een lange levensduur hebben en het beste door de periodieke tracing garbage collector opgeruimd kunnen worden.

```
1  void* new(Type) {
2      inst = malloc(sizeof(int)+sizeof(Type))
3      inst->ref_count = 1
4      ((Type)inst)->init()
5      return inst
6
7  }
8
9  void* reference(ptr) {
10     ptr->ref_count += 1
11     return ptr
12 }
13
14 void dereference(ptr) {
15     ptr->ref_count -= 1
16     if ptr->ref_count == 0
17         ptr->deinit()
18         delete ptr
19 }
20
21 void clear() {
22     foreach instance in Heap do
23         instance->is_reachable = false
24 }
25
26 void mark(node) {
27     foreach ref in node do
28         ref->is_reachable = true
29         mark(ref)
30 }
31
32 void sweep() {
33     foreach instance in Heap do
34         if (!instance->is_reachable)
35             free(instance)
36 }
37
38 void invoke_GC() {
39     pause_all_threads()
40
41     clear()
42     mark(Root)
43     sweep()
44
```

```
45      reseume_all_threads()
46  }
```

Met een aangepaste versie van dit algoritme kunnen we wel degelijk ook zekerheid bieden over wanneer de deconstructor wordt aangeroepen. Een groot nadeel van deze aanpassing van het algoritme is dat telkens wanneer we de reference-count van een instantie verlagen we de periodieke garbage collector moeten starten, en dit is mogelijk veel frequenter dan een losse periodieke garbage collector.

```
1   void* new(Type)
2
3   void* reference(ptr)
4
5   void dereference(ptr) {
6       ptr->ref_count -= 1
7       if ptr->ref_count == 0
8           ptr->deinit()
9           delete ptr
10      else
11          invoke_GC()
12  }
13
14  void clear() {
15      foreach instance in Heap do
16          instance->is_reachable = false
17  }
18
19  void mark(node) {
20      foreach ref in node do
21          (*ref)->is_reachable = true
22          mark(*ref)
23  }
24
25  void sweep() {
26      foreach instance in Heap do
27          if !instance->is_reachable
28              delete instance
29  }
30
31  void invoke_GC() {
32      pause_all_threads()
33
34      clear()
35      mark(Root)
36      sweep()
37
38      reseume_all_threads()
39  }
```

Voordelen:

- Is in staat om cyclische referenties op te ruimen.

- Frequentie waarmee de tracing-collector aangeroepen wordt kan drastisch worden gereduceerd.
- Kan m.b.v. een kleine aanpassing zekerheid bieden over moment waarop de deconstructor wordt aangeroepen.

Nadelen:

- Zekerheid van moment van aanroepen deconstructor is zeer kostbaar.

4.1.5 Stop-and-copy

Dit algoritme maakt onderscheid tussen twee geheugen gebieden, de *tospace* en de $fromspace$. Wanneer de garbage collector wordt uitgevoerd kopiëren we alle bereikbare instanties van de $fromspace$ naar de *tospace* en draaien dan de rollen om ($fromspace$ wordt *tospace* en vice versa). Op deze manier blijft alle garbage achter in de $fromspace$ en kan het geheugen worden hergebruikt. Doordat garbage niet gekopieerd wordt maken we het geheugen compacter en gaan we fragmentatie tegen. Het is ook mogelijk om het geheugen in verschillende delen op te delen die ieder een $from-$ en *tospace* bevatten zodat de garbage collector incrementeel gebruikt kan worden. Dit houdt in dat we tijdens het uitvoeren van de garbage collector een gebied behandelen i.p.v. het gehele geheugen en de volgende keer dat de garbage collector uitgevoerd wordt behandelen we een ander gebied. Een beschrijving van dit algoritme is te vinden in [10] en hier [12] in de vorm van een generational garbage collector. Om te voorkomen dat instanties met meerdere referenties tijdens het uitvoeren van de garbage collector gedupliceerd worden is het noodzakelijk om elke instantie een extra veld te geven die verwijst naar de positie waar de gekopieerde instantie zich bevindt. Als een instantie nog niet gekopieerd is, is dit veld gelijk aan $NULL$.

```
1  new_copy(Type, old, space) {
2      inst = malloc_in(space, sizeof(void*)+sizeof(Type))
3      inst->copy = NULL
4      memcpy(old+sizeof(void*), inst+sizeof(void*), sizeof(Type))
5      return inst
6  }
7
8  copy(node) {
9      foreach ref in node do
10         if (*ref)->copy != NULL
11             *ref = (*ref)->copy
12         else
13             new_inst = new_copy(TypeOf(ref), *ref, tospace)
14             (*ref)->copy = new_inst
15             *ref         = new_inst
16             copy(new_inst)
17 }
18
```

```
19 invoke_GC() {
20     pause_all_threads()
21
22     copy(Root)
23     swap(tospace, fromspace)
24
25     reseume_all_threads()
26 }
```

malloc_in() alloceert geheugen in een gegeven gebied. Tijdens een garbage collection stap is dit altijd de *tospace*, in het geval van een instantiatie in de code zal dit de $fromspace$ zijn omdat hier alle huidige instanties zich in bevinden. Wanneer een instantie al eerder is gekopieerd zal zijn *copy* veld ongelijk zijn aan $NULL$ en kunnen we simpelweg het adres dat zich in dit veld bevind kopiëren.

Voordelen:

- Is in staat om cyclische referenties op te ruimen.
- Defragementeerd geheugen tijdens elke collect stap.

Nadelen:

- Geen zekerheid over het moment van aanroepen deconstructor.
- Kopiëren heap kan kostbaar zijn.
- Vereist een stop-the-world.

4.1.6 Generational garbage collector

Generational garbage collectors maken gebruik van het feit dat de meeste objecten een korte levensduur hebben. Het geheugen wordt opgedeeld in generaties. Wanneer een object gealloceerd wordt plaatsen we hem in de jongste generatie en geven we hem een teller. Als de garbage collector een instantie als bereikbaar markeert verhogen we deze teller en wanneer deze hoog genoeg is verplaatsen we de instantie naar een oudere generatie. De garbage collector zal vaker uitgevoerd worden op jonge generaties en minder vaak op de oudere. Dit algoritme reduceert het aantal instanties dat we beschouwen tijdens het uitvoeren van de garbage collector en een garbage sweep kost zo minder tijd. Een incrementeele garbage collection algoritme wordt op de verschillende generaties toegepast. Dit zou bijvoorbeeld het stop-and-copy algoritme kunnen zijn zoals beschreven in [12, 8].

```
1 Generations = [
2     {tospace = reserve(), fromspace = reserve(), next =
          Generations[1]},
3     {tospace = reserve(), fromspace = reserve(), next =
          Generations[2]},
```

```
4       {tospace = reserve(), fromspace = reserve(), next =
            Generations[2]} ]
5
6   cur_generation = 0
7
8   new_copy(Type, old, gen) {
9       inst = malloc_in(gen->next->tospace, sizeof(void*)+sizeof(
            Type))
10      inst->copy = NULL
11      memcpy(old+sizeof(void*), inst+sizeof(void*), sizeof(Type))
12      return inst
13  }
14
15  copy_generation(gen, node) {
16      foreach ref in node do
17          if in_space(gen->fromspace, *ref)
18              if (*ref)->copy != NULL
19                  *ref = (*ref)->copy
20              else
21                  new_inst = new_copy(TypeOf(ref), *ref, gen)
22                  (*ref)->copy = new_inst
23                  *ref         = new_inst
24                  copy_generation(gen, new_inst)
25          else
26              copy_generation(gen, *ref)
27  }
28
29
30  generation_GC(gen) {
31      copy_generation(gen, Root)
32      swap(gen->tospace, gen->fromspace)
33  }
34
35  invoke_GC() {
36      pause_all_threads()
37
38      if cur_generation == 0
39          generation_GC(Generations[0])
40      if cur_generation == 1
41          generation_GC(Generations[0])
42      if cur_generation == 2
43          generation_GC(Generations[1])
44      if cur_generation == 3
45          generation_GC(Generations[0])
46      if cur_generation == 4
47          generation_GC(Generations[1])
48      if cur_generation == 5
49          generation_GC(Generations[0])
50      if cur_generation == 6
51          generation_GC(Generations[2])
52
53      reseume_all_threads()
54
55      cur_generation += 1
```

```
56      cur_generation %= 7
57  }
```

Telkens wanneer we de garbage-collector aanroepen zullen we enkel een segment van de heap beschouwen. Wanneer instanties een garbage-collect stap overleven verplaatsen we ze naar de oudere generatie, tenzij ze tot de oudste generatie behoren, dan blijven ze binnen deze generatie. Tevens zullen we jongere generaties vaker beschouwen dan oudere.

Voordelen:

- Is in staat om cyclische referenties op te ruimen.
- Defragementeert op den duur het volledige geheugen.
- Garbage-collection kan worden opgedeeld in kleinere stappen.
- Houd rekening met leeftijd van instanties.

Nadelen:

- Geen zekerheid over het moment van aanroepen deconstructor.
- Kopiëren heap kan kostbaar zijn.

4.1.7 On-the-fly garbage collector

De on-the-fly garbage collector gebruikt een algoritme waarmee de garbage collector los van de rest van het programma wordt uitgevoerd en zo dus een stop-the-world voorkomt[11, 9, 10]. Dit wordt gedaan door het programma als twee delen te zien, de *mutator* en *collector*. De *collector* herhaalt telkens de volgende stappen.

- Kleur recursief alle bereikbare nodes zwart totdat het aantal zwarte nodes niet langer toeneemt.
- Doorloop alle nodes en geef alle witte nodes vrij door ze aan de $free-list$ toe te voegen. Kleur alle zwarte nodes wit.

De *mutator* zal telkens wanneer we een nieuwe instantie aanmaken een node uit de $free-list$ pakken en deze zwart kleuren.

```
1  struct Node {
2      kleur      = Wit
3      members[]  = NULL
4  }
5
6  Node   gc_memory[MEMORY_SIZE/NODE_SIZE]
7  Node*  gc_freelist = &gc_memory[0]
8  Node*  gc_root     = &gc_memory[1]
```

```
 9
10 Node* mutator_new()
11 {
12     return gc_freelist.pop_front()
13 }
14
15 void mutator_assign(Node* parent, int member, Node* value)
16 {
17     parent->members[member] = value
18     value->kleur = Zwart
19 }
20
21 void collector_mark()
22 {
23     int aantal_zwart = 1
24     int vorig_aantal_zwart = 0
25
26     gc_root->kleur = Zwart
27     gc_freelist->kleur = Zwart
28
29     while (aantal_zwart > vorig_aantal_zwart)
30     {
31         vorig_aantal_zwart = aantal_zwart
32         aantal_zwart = 0
33
34         for node in gc_memory
35         {
36             if (node->kleur == Zwart)
37             {
38                 for member in node->members
39                 {
40                     member->kleur = Zwart
41                 }
42             }
43         }
44
45         for node in gc_memory
46         {
47             if (node->kleur == Zwart)
48             {
49                 aantal_zwart += 1
50             }
51         }
52     }
53 }
54
55 void collector_collect()
56 {
57     for node in gc_memory
58     {
59         if (node->kleur == Wit)
60         {
61             gc_freelist->push_back(node)
62         }
```

```
63          else
64          {
65              node->kleur = Wit
66          }
67      }
68  }
69
70  void collector_main()
71  {
72      while (true)
73      {
74          collector_mark()
75          collector_collect()
76      }
77  }
```

Omdat de $free - list$ door beide processen gebruikt wordt dienen we deze van een locking mechanisme te voorzien om te voorkomen dat de lijst corrupt wordt wanneer beiden processen hem veranderen. Door de $free - list$ ook als een root node te zien voorkomen we dat nodes meerdere malen vrijgegeven worden. Want nodes die zich in de $free - list$ bevinden worden de eerstvolgende keer dat $collector\_mark()$ aangeroepen wordt zwart gekleurd. Om te voorkomen dat de *collector* een node die door de *mutator* uit de $free - list$ verwijdert is en nog niet is zwart gekleurd onmiddellijk weer vrijgeeft, dient de *mutator* deze zwart te kleuren voordat deze uit de $free - list$ verwijdert wordt. Dit geldt ook voor instanties die tijdens een toekenning tijdelijk onbereikbaar zijn. Het door de *mutator* expliciet zwart kleuren gebeurd in de $mutator\_assign()$ functie.

Omdat de *collector* een groot deel van zijn tijd besteedt aan het zwart kleuren van nodes die gewoon bereikbaar zijn is er een uitbreiding op dit algoritme die gebruik maakt van drie kleuren[9]: zwart voor bereikbare nodes, wit voor nodes die onbereikbaar zijn en grijs voor nodes die mogelijk onbereikbaar geworden zijn. Telkens wanneer de *mutator* een referentie verandert maken we de oude referentie grijs. De *collector* herhaalt telkens de volgende stappen:

- Kleur alle root nodes die grijs zijn wit.
- Kleur recursief voor alle root nodes die wit zijn al hun kinderen wit.
- Kleur recursief alle bereikbare nodes die wit zijn zwart.
- Doorloop alle nodes en voeg alle witte nodes toe aan de free list.

Het voordeel van deze aangepaste versie van het algoritme is dat wanneer we een referentie naar een instantie verbreken deze grijs gekleurd zal worden. Wanneer we nu ergens anders nog een bereikbare referentie naar deze instantie hebben hoeven we voor deze node niet recursief alle subnodes opnieuw te kleuren.

```
1  void mutator_assign(Node* parent, int member, Node* value)
2  {
3      parent->members[member]->kleur = Grijs
4      parent->members[member] = value
5  }
6
7  void collector_witten
8  {
9      for node in gc_memory
10     {
11         if (node->kleur == Grijs)
12         {
13             node->kleur = Wit
14         }
15     }
16
17     klaar = false
18
19     while (!klaar)
20     {
21         klaar = true
22         for node in gc_memory
23         {
24             if (node->kleur == Wit)
25             {
26                 for member in node->members
27                 {
28                     if (member->kleur == Zwart)
29                     {
30                         klaar = false
31                         member->kleur = Wit
32                     }
33                 }
34             }
35         }
36     }
37 }
38
39 void collector_mark()
40 {
41     int aantal_zwart = 1
42     int vorig_aantal_zwart = 0
43
44     gc_root->kleur = Zwart
45     gc_freelist->kleur = Zwart
46
47     while (aantal_zwart > vorig_aantal_zwart)
48     {
49         vorig_aantal_zwart = aantal_zwart
50         aantal_zwart = 0
51
52         for node in gc_memory
53         {
54             if (node->kleur != Wit)
```

```
55              {
56                  for member in node->members
57                  {
58                      if (member->kleur == Wit)
59                      {
60                          member->kleur = Zwart
61                      }
62                  }
63              }
64          }
65
66          for node in gc_memory
67          {
68              if (node->kleur != Zwart)
69              {
70                  aantal_zwart += 1
71              }
72          }
73      }
74  }
75
76  void collector_collect()
77  {
78      for node in gc_memory
79      {
80          if (node->kleur == Wit)
81          {
82              gc_freelist.push_back(node)
83              node->kleur = Zwart
84          }
85      }
86  }
87
88  void collector_main()
89  {
90      while (true)
91      {
92          collector_witten()
93          collector_mark()
94          collector_collect()
95      }
96  }
```

In het bovenstaande algoritme worden niet langer alle nodes na het doorlopen van een garbage-collection cycle wit gekleurd. In plaats daarvan kleuren we alle nodes die plaats hebben gemaakt voor een andere node in een assignment grijs omdat dit de enige nodes zijn die mogelijk onbereikbaar zijn geworden.

Voordelen:

- GC vereist geen stop-the-world.

Nadelen:

- Geen controle over moment waarop garbage vrijgeven wordt.

4.1.8 Ownership based

Alle instanties hebben één referentie die als eigenaar van de instantie wordt beschouwd. Deze referentie telt niet mee in de reference-count van de instantie, alle overige referenties wel. Wanneer de eigenaar van een instantie out-of-scope raakt zal de instantie worden opgeruimd. Wanneer na het opruimen de reference-count voor de niet-eigenaarsreferenties groter is dan 0 resulteert dit in een run-time error. Een speciaal klasse referentie type geeft aan of een referentie de eigenaar is van een instantie en het eigenaarschap kan worden doorgegeven met een speciale operator. Omdat er altijd maar één eigenaar van een instantie kan zijn wordt bij het doorgeven van het eigenaarschap de bron referentie $NULL$/ongeldig gemaakt. Het is nu nog wel mogelijk om cyclische referenties te maken die niet worden opgeruimd door een object eigenaar te maken van zichzelf. Het volgende artikel laat zien hoe dit algoritme werkt en beschrijft tevens een aangepaste versie van C# die gebruik maakt van deze vorm van garbage-collection [5].

```
1  void* claim(ref) {
2      new_ref = *ref
3      *ref    = NULL
4      return new_ref
5  }
6
7  void* reference(ptr) {
8      ptr->ref_count += 1
9      return ptr
10 }
11
12 dereference(ptr) {
13     if ptr IS OWNER
14         ptr->deinit()
15         if ptr->ref_count != 0
16             ERROR 1
17         delete ptr
18     else
19         ptr->ref_count -= 1
20 }
```

De ownership based garbage collector vereist de volgende acties.

- Wanneer een niet-eigenaarsreferentie gemaakt wordt verhogen we de reference-count.
- Wanneer een niet-eigenaarsreferentie out-of-scope raakt verlagen we de reference-count.

- Na het alloceren van een nieuwe instantie moeten we deze in een referentie van een speciaal eigenaarstype bewaren.
- Wanneer we het eigenaarschap doorgeven aan een andere referentie met dit speciale eigenaarstype wordt de voormalige eigenaar ongeldig gemaakt.
- Als een referentie van een speciaal eigenaarstype out-of-scope raakt ruimen we de instantie op. Indien na het opruimen de reference-count van de instantie niet gelijk is aan 0 geven we een run-time error.

Door een instantie en zijn mogelijke members eerst vrij te geven alvorens de reference-count te controleren is het mogelijk om een instantie naar zichzelf te laten verwijzen, mits deze verwijzing geen eigenaarsreferentie is.

Voordelen:

- Dit algoritme vereist geen stop-the-world of parallel garbage-collection thread.
- Programmeur heeft controle over vernietiging van instanties.

Nadelen:

- Memory-leaks vanwege cyclische referenties zijn nog steeds mogelijk.
- Programmeur moet er voor zorgen dat wanneer een eigenaarsreferentie out-of-scope geraakt er geen niet-eigenaarsreferenties meer zijn.

Conclusie

Elke garbage-collector heeft tekortkomingen waardoor de eisen die voor cgt gesteld zijn niet gehaald worden. Reference-counting kan geen cyclische referenties opruimen. Mark and Sweep, stop-and-copy, en on-the-fly garbage collectors bieden geen garantie dat een deconstructor op een bekend moment wordt uitgevoerd en een ownership based garbage-collector kan in gevallen waarin objecten zichzelf bezitten nog steeds voor memory leaks zorgen. Daarom bouwen we in de volgende sectie door aan het ownership based algoritme om zo deze tekortkoming te verhelpen.

4.2 Nieuwe oplossingen

4.2.1 Dynamic ownership

Door alle instanties op een even adres in het geheugen te plaatsen kunnen we het adres gebruiken om het eigenaarschap van een instantie te coderen. De referentie die de eigenaar is van een instantie zal de least-significant-bit

van het adres hoog hebben (gelijk aan 1). Alle normale referenties hebben deze bit laag (gelijk aan 0). Voordat we het adres van een instantie gebruiken dienen we altijd de least-significant-bit te wissen, en wanneer we het eigenaarschap doorgeven wordt in de bron deze bit gewist en in het doel hoog gemaakt. Het is nu niet langer nodig een speciaal type voor eigenaars te introduceren omdat we eigenaars- en niet-eigenaarsreferenties aan de hand van het adres kunnen onderscheiden. Wanneer we een nieuwe instantie initialiseren zal *nieuw*/*nieuwe* een eigenaarsreferentie opleveren. Het eigenaarschap kunnen we daarna m.b.v. een speciale operator doorgeven. Tevens is er een speciale operator om te controleren of een referentie de eigenaar is van de instantie waarnaar deze verwijst. Dit doen we door de least-significant-bit van het adres te controleren.

```
1  access(ptr) {
2      return ptr & 0xFFFFFFFE
3  }
4
5  new(Type) {
6      inst = malloc(1+sizeof(int)+sizeof(Type))
7
8      if inst&0x00000001 == 0
9          inst += 1
10
11     access(inst)->ref_count = 0
12
13     return inst
14 }
15
16 claim(ref) {
17     if (*ref)&0x00000001 == 0
18         ERROR 1
19
20     new_ref = (*ref)
21     *ref    = reference(*ref)
22
23     return new_ref
24 }
25
26 reference(ptr) {
27     ptr = access(ptr)
28     ptr->ref_count += 1
29     return ptr
30 }
31
32 dereference(ptr) {
33     if ptr&0x00000001 != 0
34         ptr->deinit()
35         if ptr->ref_count != 0
36             ERROR 1
37         delete ptr
38     else
39         ptr->ref_count -= 1
```

```
40 }
```

Wanneer we een instantie aanmaken verkrijgen we een oneven adres welke aanduidt dat wij de eigenaar zijn van de desbetreffende instantie. Telkens wanneer we dit adres kopiëren en elders gebruiken wissen we de least-significant-bit en verhogen we de reference-count. Wanneer een referentie out-of-scope raakt controleren we de least-significant-bit. Als deze 0 is verlagen we de reference-count, indien deze bit 1 is (en de eigenaar dus out-of-scope gaat) geven we de instantie vrij. Als de reference-count dan nog steeds ongelijk is aan 0 dan resulteert dit in een runtime-error. Wanneer we het eigenaarschap van een instantie doorgeven en de huidige referentie is niet de eigenaar dan resulteert dat ook in een runtime-error want het is onmogelijk om twee referenties te hebben die beide eigenaar zijn van dezelfde instantie. Het bovenstaande algoritme gaat uit van een architectuur met 32bits adressen. De *claim*() functie wordt hier gebruikt om het eigenaarschap door te geven, dit doen we door de oorspronkelijke eigenaar tot een niet-eigenaarsreferentie om te zetten en de eigenaarsreferentie als return-value door te geven.

```
1  Sjabloon Foo
2  {
3      bar : Tekst.
4  }
5
6  print(foo : Foo) : Geen {
7      ~scherm..schrijf(foo->bar).
8  }
9
10 start(pr:[Tekst]) : B {
11     foo : Foo = nieuwe Foo().
12
13     foo->bar = "Hello World".
14
15     print(foo).
16
17     verlaat ja.
18 }
```

Bovenstaande cgt programma zal er als volgt uitzien wanneer we de garbage collector invokaties expliciet toevoegen.

```
1  Sjabloon Foo
2  {
3      bar : Tekst.
4  }
5
6  print(foo : Foo) : Geen {
7      ~scherm..schrijf(access(foo)->bar).
8      dereference(foo).
9  }
10
11 start(pr:[Tekst]) : B {
```

```
12      foo : Foo = new(Foo).
13
14      access(foo)->bar = "Hello World".
15
16      print(reference(foo)).
17
18      dereference(foo).
19      verlaat ja.
20  }
```

Met deze aanpassing reduceren we het aantal aanpassingen aan de syntaxis van de taal die nodig zijn om hem geschikt te maken voor een ownership based garbage collector.

4.2.2 Dynamic ownership met periodieke tracing collector

In sectie 4.1.8 is duidelijk geworden dat een standaard ownership based collector nog steeds niet in staat is alle mogelijke cyclische referenties te detecteren en op te ruimen. Om dit op te lossen kunnen we net als bij standaard reference-counting een periodieke tracing collector toevoegen. In deze sectie combineren we de dynamic ownership based collector uit sectie 4.2.1 met een Mark and Sweep tracing collector, maar het is ook mogelijk om een andere tracing collector te gebruiken. Wanneer de tracing collector onbereikbare instanties lokaliseert zullen deze niet op de standaard manier worden opgeruimd maar zal de collector op zoek gaan naar een punt in de referentiecykel om te doorbreken. Om dit mogelijk te maken moeten we voor elke instantie een extra veld bijhouden met daarin het adres naar de node die de eigenaarsreferentie bevat. Als de eigenaarsreferentie een sjabloon variabele is wordt hier dus het adres van deze sjabloon instantie bewaard. Indien een lokale of globale variabele de eigenaar is, dan is dit veld leeg. Als de tracing collector een onbereikbare instantie vindt markeren we deze instantie als bereikbaar en dan beschouwen we zijn eigenaar die ook als bereikbaar wordt gemarkeerd. Dit proces herhalen we tot we een instantie bereiken die als bereikbaar is gemarkeerd. Dit betekent dat we een instantie hebben gelokaliseerd die onderdeel is van de cykel. We laten nu kunstmatig de member variabele die de eigenaarsreferentie naar de vorige instantie in de cykel bevat out-of-scope raken. Waardoor de cykel wordt doorbroken en de ownership based collector begint met opruimen. Omdat er onbereikbare objecten zijn gemarkeerd kunnen we geen verdere instanties meer opruimen zonder eerst de tracing collector opnieuw te starten. De tracing collector kan dus maximaal een cykel opruimen per aanroep opruimen.

```
1  access(ptr) {
2      return ptr & 0xFFFFFFFE
3  }
4
5  new(Type) {
6      inst = malloc(1+sizeof(int)+sizeof(Type))
```

```
7
8       if inst&0x00000001 == 0
9           inst += 1
10
11      access(inst)->ref_count = 0
12      access(inst)->owner     = NULL
13
14      return inst
15  }
16
17  claim(ref, instance) {
18      if (*ref&0x00000001 == 0)
19          ERROR 1
20
21      new_ref = (*ref)
22      (*ref)  = reference(*ref)
23
24      access(new_ref)->owner = instance
25      return new_ref
26  }
27
28  reference(ptr) {
29      ptr = access(ptr)
30      ptr->ref_count += 1
31      return ptr
32  }
33
34  dereference(ptr) {
35      if (ptr&0x00000001 != 0)
36          ptr->deinit()
37          if (ptr->ref_count != 0)
38              ERROR 1
39          free(ptr)
40      else
41          ptr->ref_count -= 1
42  }
43
44  break_cycle(instance)
45  {
46      if (instance->is_reachable)
47      {
48          owner_ref = find_member_by_contents(instance->owner)
49          *(owner_ref) = NULL
50          dereference(instance->owner)
51          return
52      }
53
54      instance->is_reachable = true
55      break_cycle(instance->owner)
56  }
57
58  clear() {
59      foreach instance in Heap do
60          instance->is_reachable = false
```

```
61 }
62
63 mark(node) {
64     foreach ref in node do
65         if (ref->is_reachable != true)
66             ref->is_reachable = true
67             mark(ref)
68 }
69
70 sweep() {
71     foreach instance in Heap do
72         if (instance->is_reachable == false)
73             break_cycle(instance)
74             return
75 }
76
77 invoke_GC() {
78     pause_all_threads()
79
80     clear()
81     mark(Root)
82     sweep()
83
84     reseume_all_threads()
85 }
```

Wanneer de tracing collector een onbereikbare instantie vindt wordt *break_cycle* aangeroepen. Deze functie gaat op zoek naar een instantie die onderdeel is van de cykel want het kan zo zijn dat een onbereikbare instantie niet deel uitmaakt van de cykel maar enkel een member is van een instantie die deel uitmaakt van de cykel. Daarna verbreken we de link tussen twee instanties in deze cykel door de eigenaar van de gevonden instantie out-of-scope te halen. Assignments die gebruik maken van de *onteigen*() operatie moeten nu dus ook aangeven wie de nieuwe eigenaar wordt. Hieronder staan enkele voorbeelden van hoe bovenstaande functies worden toegevoegd aan cgt code.

```
1  foo : Foo = nieuwe Foo().
2
3  # globale/lokale toekenning
4  lokale_foo : Foo = foo->onteigen().
5  # wordt
6  lokale_foo : Foo = claim(foo, niets).
7
8  # voor members
9  ik->member_foo = foo->onteigen().
10 # wordt
11 ik->member_foo = claim(foo, ik).
12
13 # voor parameters
14 mijn_funct(foo->onteigen()).
15 # wordt
16 mijn_funct(claim(foo, niets)).
```

Met deze garbage collector wordt het nu mogelijk om geen gebruik te maken van de ownership-based collector door het object eigenaar van zichzelf te maken. Dit komt omdat de tracing collector de bereikbaarheid niet alleen op eigenaarsreferenties baseert maar ook niet-eigenaarsreferenties beschouwt.

Er kan zich nu nog wel een runtime error voordoen wanneer we in de deconstructor gebruik maken van een member wiens inhoud door de tracing-collector kunstmatig out-of-scope is gehaald. Daarnaast is de instantie wiens link we doorbreken om de cykel op te ruimen ook niet op voorhand te voorspellen maar afhankelijk van de volgorde waarin instanties zich in het geheugen bevinden.

4.3 Conclusie

Met de uitbreidingen op de ownership based collector zoals besproken in secties 4.2.1 en 4.2.2 hebben we een geschikte garbage-collector voor cgt geformuleerd. In hoofdstuk 5 worden de benodigde aanpassingen aan de taal besproken. De volgende versie van cgt zal gebruik maken van de ownership based tracing hybrid zoals besproken in 4.2.2, hoewel het tracing gedeelte in de rest van deze scriptie buiten beschouwing wordt gelaten.

Hoofdstuk 5

Consequenties voor de taal

In dit hoofdstuk bespreken we de veranderingen aan cgt die nodig zijn om de nieuwe garbage collector zoals besproken in sectie 4.2.2 te verwezenlijken. In sectie 5.1 beschouwen we enkel de taal. Sectie 5.3 gaat in op de consequenties voor het runtime systeem.

5.1 Uitbreiding taal

- *nieuw/nieuwe* resulteert in een eigenaarsreferentie.
- De $\$referentie$ operator wist de bron referentie en legt een eigenaarsreferentie op de stapel. Wanneer de bron referentie geen eigenaar is van de instantie geeft dit een runtime-error. Deze operator kan gebruikt worden om het eigenaarschap aan een functie/operatie parameter door te geven, of om in combinatie met de $ = operator zowel het eigenaarschap door te geven en de bronreferentie te wissen.
- Wanneer we het eigenaarschap m.b.v. een toekenning willen doorgeven gebruiken we de $ = operator. De bron referentie wordt nu een niet-eigenaarsreferentie in tegenstelling tot de $\$referentie$ operator die de bronreferentie wist.
- De $ref \rightarrow is\_eigenaar()$ operator verlaat een boolean waarmee gecontroleerd kan worden of de huidige referentie de eigenaar is van een instantie.
- Wanneer een referentie die eigenaar is van een instantie, out-of-scope raakt wordt de instantie opgeruimd en de deconstructor aangeroepen.
- Wanneer een instantie wordt opgeruimd en na het uitvoeren van de deconstructor de reference-count niet 0 is geven we een runtime-error.
- Wanneer we een referentie kopiëren zonder gebruik te maken van speciale operatoren veranderd deze in een niet-eigenaarsreferentie.

- Wanneer een eigenaarsreferentie niet bewaard wordt (b.v. door een return value te negeren) wordt deze meteen opgeruimd.
- De impliciete *ik* parameter die alle operaties automatisch hebben is altijd een niet-eigenaarsreferentie naar de instantie.
- Wanneer er geen bereikbare eigenaars *en* niet-eigenaarsreferenties naar een instantie verwijzen ruimt de tracing garbage collector deze instantie op.
- Instanties die door de tracing collector opgeruimd worden moeten in hun deconstructor rekening houden met het feit dat sommige members mogelijk niet langer geldig zijn.

De volgende regels worden aan de grammatica toegevoegd of uitgebreid. Voor de volledige (originele) grammatica zie appendix A.

Toekenning	=	Symbool "$ =" Symbool
	|	Symbool ToeOp Symbool
ToeOp	=	"+="
	|	"-="
	|	"*="
	|	"/="
	|	"="
Waarde	=	Const
	|	WaardeSym
	|	Onteigen
	|	Nieuwe
	|	NieuweRij
Onteigen	=	"$" Symbool

5.2 Voorbeeld

```
1  # voorbeeld cyclische referentie
2  #  met nieuwe garbage-collector geeft
3  #  deze code niet langer een memory-leak.
4
5  Sjabloon Foo
6  {
7      ref : Foo.
8  }
9
10 start(pr:[Tekst]) : B {
11     # maak een instantie van Foo
12     #  foo is nu de eigenaar van de instantie.
13     foo : Foo $= nieuwe Foo().
14
15     # zorg dat foo een referentie
16     #  naar zichzelf bevat
17     foo->ref = foo.
18
19     # klaar
20     # foo raakt out-of-scope
21     # instantie wordt gedeinitialiseerd en
22     # foo->ref raakt ook out-of-scope dus
23     # reference count foo = 0
24     verlaat ja.
25 }
```

We beschouwen nogmaals het voorbeeld uit sectie 4.1.1 dat werd gebruikt om te laten zien dat m.b.v. reference-counting het mogelijk is om memory-leaks te veroorzaken. Met de nieuwe garbage-collector is dit niet langer het geval en wordt de instantie van *Foo* netjes opgeruimd. Het is nog wel mogelijk om cyclische referenties te maken die niet opgeruimd worden door de ownership based garbage collector. Een voorbeeld wordt hieronder gegeven.

```
1  # voorbeeld memoryleak
2
3  Sjabloon Foo
4  {
5      ref : Foo.
6  }
7
8  start(pr:[Tekst]) : B {
9      # maak een instantie van Foo
10     #  foo is nu de eigenaar van de instantie.
11     foo : Foo $= nieuwe Foo().
12
13     # zorg dat foo->ref de instantie bezit
14     #  foo is nu niet langer de eigenaar
15     foo->ref $= foo.
16
17     # klaar
```

```
18      # foo raakt out-of-scope maar de eigenaar niet
19      # reference count is nu 0
20      verlaat ja.
21  }
```

De tracing collector zal nu de instantie opruimen omdat deze niet langer bereikbaar is. Omdat de tracing collector geen onderscheid maakt tussen eigenaars en niet-eigenaarsreferenties kunnen we de ownership based collector buitenspel zetten door een instantie zichzelf te laten bezitten. Nu zal de tracing collector de instantie opruimen zo gauw deze niet langer bereikbaar is.

```
1  Sjabloon EigenBaas
2  {
3      ikke : EigenBaas.
4  }
5
6  foo : EigenBaas = nieuwe EigenBaas().
7  foo->ikke $= foo.
```

5.3 Consequenties voor het runtime systeem en compiler

De compiler en virtuele machine zullen met de volgende operaties rekening moeten houden met de nieuwe garbage-collector.

Adres op de stapel leggen. Hier nemen we het adres naar de instantie zoals een referentie hem bevat en wissen we de least-significant-bit en verhogen de reference-count.

Adres van stapel naar referentie verplaatsen. Het adres zoals deze zich op de stapel bevindt wordt rechtstreeks naar de doelreferentie verplaatst zonder de reference-count te veranderen.

Instantie opruimen. Allereerst voeren we de deconstructor uit en daarna behandelen we alle member variabelen door ze out-of-scope te laten gaan (ofwel reference count verlagen ofwel opruimen). Nadat we dit allemaal gedaan hebben controleren we of de reference count van de instantie gelijk is aan 0, is dit niet het geval dan geven we een runtime-error omdat er nog steeds referenties naar deze instantie te vinden zijn terwijl deze niet langer bestaat.

Adres op stapel wordt verworpen. Als least-significant-bit van het adres laag (gelijk 0) is wordt de reference count met 1 verlaagd. Wanneer de least-significant-bit hoog is wordt de instantie opgeruimd.

Eigenaarschap doorgeven. Allereerst wordt gecontroleerd of de bron referentie wel een adres heeft waarvan de least-significant-bit hoog is. Als dit niet het geval is resulteert dit in een runtime-error. Daarna maken we de least-significant-bit in de bronreferentie 0 zodat deze niet langer de eigenaar is van de instantie en leggen we het adres op de stapel. Het nieuwe adres op de stapel heeft wel zijn least-significant-bit hoog en is dus de nieuwe eigenaar van de instantie. Wanneer de $ operator gebruikt wordt maken we de bron referentie gelijk aan *niets*.

Referentie raakt out-of-scope. Wanneer de least-significant-bit van het adres naar de instantie laag is wordt de reference count van de instantie met 1 verlaagd. Als de least-significant-bit hoog is wordt de instantie opgeruimd.

Tracing collector. Daarnaast dienen we een tracing collector toe te voegen. Hiervoor moeten we bijhouden welke referenties op de stapel naar een instantie verwijzen zodat deze als bereikbaar gemarkeerd kunnen worden. De eerstvolgende versie van cgt zal hiervoor gebruik maken van Mark and Sweep.

5.3.1 Nieuwe codering sjabloon instanties in geheugen

In sectie 2.10.1 wordt beschreven hoe sjabloon instanties in het geheugen bestaan. De nieuwe garbage collector vereist ook aanpassingen in dit gebied. De codering van deze instanties ziet er nu als volgt uit. Het reference-count veld wordt nu een reference-count enkel voor niet-eigenaarsreferenties. Er wordt een nieuw veld toegevoegd dat verwijst naar de sjabloon instantie wiens member variabele de eigenaar is van de instantie. Dit veld wordt enkel gebruikt wanneer de eigenaarsreferentie een sjabloon member variabele is.

Tabel 5.2: Sjabloon Instantie Codering

Adres	Lengte	Inhoud
0h0000	4	Niet eigenaars reference-count.
0h0004	1	Id (zie Instantie Typen).
0h0005	8	Adres van eigenaarsinstantie.
0h000D	4	Positie van init operatie.
0h0011	4	Positie van deinit operatie.
0h0015	8	Adres van gegenereerde deref operatie.
0h001D	4	'geldig' element van dit sjabloon.
0h0021	n	Sjabloon elementen.

Hoofdstuk 6

Prototype

In dit hoofdstuk hebben we het over de originele versie van de taal als we over cgt spreken. De nieuwe versie die gebruik maakt van de in deze scriptie beschreven garbage-collector wordt cgt-next genoemd. Vooralsnog laten we rijen buiten beschouwing en zullen we de nieuwe garbage-collector enkel toepassen op sjabloon instanties.

6.1 Detecteren van memory leaks

De virtuele machine die de door de cgt compiler gegenereerde bytecode uitvoert is uitgebreid met een memoryleak detector. Deze uitbreiding omvat het bijhouden van een lijst met alle gealloceerde geheugenblokken. Wanneer een geheugenblok wordt vrijgegeven verwijderen we deze uit de lijst. Wanneer een programma termineert en als er zich nog niet vrijgegeven geheugenblokken in de lijst bevinden printen we een opsomming van deze blokken.

Neem bijvoorbeeld het volgende cgt programma.

```
1  >#  leak.cgt
2  #
3  # Een voorbeeld programma dat een memory leak veroorzaakt
4  # wanneer de oude garbage collector wordt gebruikt.
5  #<
6
7  Rubriek leak.
8
9  Sjabloon Node
10 {
11 vol : Node.
12
13 init(n:Node) : B
14 {
15 ik->vol = n.
16
17 verlaat ja.
```

```
18 }
19
20 opvolger(n:Node) : Geen {
21 ik->vol = n.
22 }
23 }
24
25 start(pr:[Tekst]) : B {
26 a : Node = nieuwe Node(niets).
27 b : Node = nieuwe Node(a).
28
29 a->opvolger(b).
30
31 verlaat ja.
32 }
```

Wanneer we dit programma compileren en met de nieuwe virtuele machine uitvoeren krijgen we nadat het programma is getermineerd de melding dat er nog twee niet vrijgegeven instanties van *Node* in het geheugen te vinden zijn.

6.2 Uitbreiding instructieset

De instructieset die door de virtuele machine begrepen wordt dient ook uitgebreid te worden om de garbage-collector te ondersteunen. Voor een definitie van de originele instructieset zie appendix A.3. De bestaande instructies die gebruikt worden om adressen naar/van de stapel te pushen/poppen worden aangepast zodat deze altijd eerst geconverteerd worden naar een niet-eigenaarsreferentie en daarna de reference-count wordt verhoogd.

- ADRDUB, deze instructie legt de bovenste waarde (een adres) op de stapel nogmaals op de stapel. Als het orginele adres een eigenaarsreferentie is wordt het nieuwe adres eerst omgezet naar een niet-eigenaarsreferentie.
- ELEM, met deze instructie wordt het adres van een sjabloon omgezet naar het adres van een member variabele en wordt de reference-count van de orginele instantie verlaagd. Wanneer het orginele adres een eigenaarsreferentie is zal deze instructie voor een runtime-error zorgen omdat het verlagen van de reference-count er voor zorgt dat de instantie wordt opgeruimd en het adres van de member variabele niet langer geldig is.
- NWSJA, is de instructie die een nieuwe sjabloon instantie aanmaakt. Oorspronkelijk werd er een referentie naar de nieuwe instantie op de stapel gelegd met een reference-count van 1. Nu wordt er een eigenaarsreferentie op de stapel gelegd met een reference-count van 0.

Daarnaast introduceren we enkele nieuwe instructies die nodig zijn om referenties naar sjabloon instanties op de stapel te leggen en eraf te pakken. Deze nieuwe instructies zorgen ervoor dat het eigenaarschap correct wordt doorgegeven of juist bij de oorspronkelijke referentie blijft.

Tabel 6.1: Nieuwe instructies voor sjabloon instanties

Code	Naam	Parameters	Effect
0x25	CLAIM	-	ref, adres → -, (deref(*ref), *ref = adres)
0x26	ONTEIGEN	-	ref → **ref, (*ref = onteigen(**ref))
0x27	LEG_ONTEIGEN	-	ref → onteigen(*ref)

De CLAIM instructie wordt gebruikt om een eigenaarsreferentie van de stapel af te pakken en het eigenaarschap door te geven aan een nieuwe referentie. De tweede instructie ontneemt het eigenaarschap van een referentie en legt een eigenaarsreferentie op de stapel. De LEG_ONTEIGEN instructie legt juist de niet-eigenaarsreferentie op de stapel en laat de oorspronkelijke referentie het eigenaarschap behouden. $deref()$ verlaagt de reference-count van een instantie tenzij het om een eigenaarsreferentie gaat, dan wordt de instantie vrijgegeven.

Hoofdstuk 7

Conclusies

In deze scriptie hebben we een introductie van de taal cgt geven en besproken waar geldige cgt programma's uit bestaan. Het huidige geheugen model is beschreven en de werking van de reference-counting garbage-collector verklaard. Deze garbage-collector is niet in staat om in alle situaties niet langer gebruikt geheugen vrij te geven. Een literatuurstudie van bestaande garbage-collectors resulteerde niet in een algoritme dat aan de door cgt gestelde eisen voldeed. De collectors die in staat zijn cyclische referenties te detecteren en vrij te geven zijn over het algemeen niet in staat de programmeur controle te geven over het moment van deconstructie, dit is een van de eisen die cgt aan de garbage-collector stelt. Door een combinatie van ownership based en een tracing garbage collector te definiëren lossen we het probleem met cyclische referenties op en is het mogelijk om zekerheid te krijgen over het moment waarop instanties opgeruimd worden. Hoewel het uiteindelijke algoritme de semantiek van de taal toch wezenlijk beïnvloedt voldoet deze toch het beste aan de gestelde eisen. Een belangrijk aspect van de runtime omgeving, namelijk performance is niet aan bod gekomen in deze scriptie. Het aantal handelingen dat aan het runtime systeem wordt toegevoegd maakt dit tot een logisch vervolg onderzoek.

Bibliografie

[1] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15:4:255–271, 2002.

[2] George A. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3:12:655–657, 1960.

[3] Emmanuel Briot. Gem #100: reference counting in ada – part 3: weak references. *ACM SIGAda Ada Letters*, 32:2:33–34, 2012.

[4] Alejandro D. Martínez and Rosita Wachenchauzer. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:1:31–35, 1990.

[5] Adam Dingle and David F. Bacon. Ownership where you can count on: A hybrid approach to safe explicit memory management. link http://researcher.watson.ibm.com/researcher/files/us-bacon/Dingle07Ownership.pdf, visited 27/08/2015.

[6] Peter L. Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19:9:522–526, 1976.

[7] Chin-Yang Lin and Ting-Wei Hou. A lightweight cyclic reference counting algorithm. In José E. Moreira Yeh-Ching Chung, editor, *Advances in Grid and Pervasive Computing*. Springer, First Internation Conference, GPC 2006, 2006.

[8] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for haskell. *FPCA '93 Proceedings of the conference on Functional programming languages and computer architecture*, 1993.

[9] Ben-Ari Mordechai. Algorithms for on-the-fly garbage collection. *ACN Transactions on Programming Languages and Systems (TOPLAS)*, 6:3:333–344, 1984.

[10] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[11] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21:11:522–526, 1978.

[12] Benjamin Zorn. Comparing mark-and sweep and stop-and-copy garbage collection. *Proceeding LFP '90 Proceedings of the 1990 ACM conference on LISP and functional programming*, 1990.

Appendix A

Appendix

A.1 Notatie

De invoer voor de parser generator die de cgt parser genereert. Het is al volgt genoteerd. Met # beginnen we commentaar dat tot het einde van de regel doorloopt.

: regexp .

Deze reguliere expressie wordt aan de expressie voor whitespace toegevoegd, tussen elk woord probeert de parser zoveel mogelijk van deze expressies te lezen totdat dit niet meer kan.

n : regexp .

Definieert de reguliere expressie voor een woord die later in de grammatica regels gebruikt kan worden. Karakters kunnen we met 'c' schrijven en % is de escape character '\' in C. Een set met tekens kan opgegeven worden door alle tekens binnen [en] te schrijven, als twee tekens met een - verbonden worden binnen een set worden alle tekens die qua karakters code tussen de twee tekens liggen ook toegevoegd.

cijfer : [0-9] .

Het woord cijfer kan nu het teken 0, 1, 2, 3, 4, 5, 6, 7, 8 of 9 zijn. Wanneer tekens of een set van tekens herhaaldelijk voor mag komen gebruiken we de $*$ of + operator, + betekend een herhaling van tenminste 1 keer en $*$ nul keer of meer.

getal : [0-9]+ .

Getal is een herhaling van cijfers bestaande uit minimaal 1 cijfer.

naam := regel .

Een grammatica regel, de '|' (of) operator heeft de laagste prioriteit.
Andere grammatica regels duiden we aan door gewoon de naam te schrijven en woorden (waarvan we de reguliere expressie hebben opgegeven) schrijven we door de naam met [ ] te omvatten (bv [n]). Sleutelwoorden kunnen direct in de regel geschreven worden door ze met " " te omvatten.

A.2 Grammatica

Start	=	(Rubriek Inhoud)*
Rubriek	=	”Rubriek” RubriekNaam? ”.”
RubriekNaam	=	” ”? ([n] ”..”)* [n]
Inhoud	=	Sjabloon
	|	Functie
	|	Variabel
	|	Constante
Type	=	TypeBasis
	|	TypeRij
	|	TypeFunct
TypeBasis	=	TypeIntern
	|	TypeSjabloon
TypeIntern	=	”N8” | ”N16” | ”N32” | ”N64” | ”N”
	|	”Z8” | ”Z16” | ”Z32” | ”Z64” | ”Z”
	|	”R32” | ”R64” | ”R”
	|	”B”
	|	”Teken” | ”Tekst”
	|	”Geen”
TypeSjabloon	=	RubriekNaam
TypeRij	=	”[” (Berekening ”\”)? Type ”]”
TypeFunct	=	Type ”<-” TypeFParams
TypeFParams	=	”(” TypeParams? ”)”
TypeParams	=	(TypeParam ”;”)* TypeParam
TypeParam	=	Type
Berekening	=	CondEn
	|	CondOf
CondEn	=	(CondEn ”en”)? Verglk
CondOf	=	(CondOf ”of’)? Verglk
Verglk	=	(Plus VerglkOp)? Plus
Plus	=	(Plus PlusOp)? Keer

Keer	=	(Keer KeerOp)? Comp
Comp	=	"!" Comp
	|	"(" Berekening ")"
	|	Waarde
VerglkOp	=	">=" | "<=" | ">" | "<" | "==" | "!="
PlusOp	=	"+" | "-" | "&" | "|" | "!|"
KeerOp	=	"*" | "/" | "%"
Waarde	=	Const
	|	WaardeSym
	|	Nieuwe
	|	NieuweRij
Const	=	ConstGetal | ConstR | ConstB
	|	ConstTeken | ConstTekst
	|	ConstNiets
ConstGetal	=	[g] | [hex] | [bin]
ConstR	=	[r]
ConstB	=	"ja" | "nee"
ConstTeken	=	[tkn]
ConstTekst	=	[t]
ConstNiets	=	"niets"
WaardeSym	=	Vader
	|	Symbool
Vader	=	"Vader" "->" [n] SymParamSet
Symbool	=	SymIdx
	|	SymRoep
	|	SymElem
	|	SymRefBasis
SymIdx	=	Symbool "[" Berekening "]"
SymRoep	=	Symbool SymParamSet
SymElem	=	Symbool "->" [n]
SymRefBasis	=	RubriekNaam
SymParamSet	=	"(" SymParams? ")"
SymParams	=	(Berekening ";")* Berekening
Nieuwe	=	("nieuw" | "nieuwe") Type SymParamSet

NieuweRij	=	"$" "[" RijElems? "]"
RijElems	=	(Berekening ";")* Berekening
Sjabloon	=	"Sjabloon" [n] SjabloonErf? SjabloonInh
SjabloonErf	=	"<-" "(" RubriekNaam ")"
SjabloonInh	=	"" SjabloonElem* ""
SjabloonElem	=	SjabloonVar
	|	SjabloonFunct
SjabloonVar	=	[n] ":" Type "."
SjabloonFunct	=	[n] Parameters ":" Type CodeBlok
Parameters	=	"(" ParamSet? ")"
ParamSet	=	(Param ";")* Param
Param	=	[n] ":" Type
Functie	=	[n] Parameters ":" Type CodeBlok
Variabel	=	[n] ":" Type "=" Berekening "."
Constante	=	[n] ":" Type ":=" Berekening "."
CodeBlok	=	"" Code* ""
Code	=	VarLokaal
	|	Toekenning
	|	Aanroep
	|	Als
	|	Zolang
	|	Alle
	|	Verlaat
	|	"opnieuw" "."
	|	"ontspring" "."
VarLokaal	=	[n] ":" Type ("=" Berekening)? "."
Toekenning	=	Symbool ToekenOp Berekening "."
ToekenOp	=	"=" | "+=" | "-=" | "*=" | "/="
Aanroep	=	Vader "."
	=	Symbool "."
Als	=	"als" "(" Berekening ")" CodeBlok Anders?
Anders	=	"anders" Als
	|	"anders" CodeBlok
Zolang	=	"zolang" "(" Berekening ")" CodeBlok

Alle	=	”alle” ”(” Toekenning ”;” Berekening ”;” Toekenning ”)” CodeBlok
Verlaat	=	”verlaat” Berekening? ”.”

A.3 Instructieset

Alle instructies en hun code (8bits).

Parameter aanduiding.

- a: Adres parameter (64bits).
- c8: 8bits constante.
- c16: 16bits constante.
- c32: 32bits constante.
- c64: 64bits constante.
- w8: 8bits waarde.
- w16: 16bits waarde.
- w32: 32bits waarde.
- w64: 64bits waarde.
- test: test operatie, zie tabel Test Operaties.

Tabel A.2: Test Operaties

Waarde	Operatie
0x00	Groter.
0x01	Kleiner.
0x10	Groter of gelijk.
0x11	Kleiner of gelijk.

Tabel A.3: CGT VM Instructies

Opcode	Naam	Parameters	Stapel (voor/na)
0x00	LEEG	-	$- \rightarrow -$
0h07	DUB32	-	$c32 \rightarrow c32, c32$
0x08	LEGFRM	c32	$- \rightarrow FRM + c32$
0x09	ADRDUB	-	$a \rightarrow a, a$
0x0A	ELEM	c32 (n)	$a \rightarrow a + n$
0x0B	OP	c32 (n)	$a \rightarrow a + n, a$
0x0C	INDEX	-	$a, c32(n) \rightarrow *(a + 9) + n$
0x0D	LENGTE	-	$a(rij) \rightarrow *(a + 5)$
0x0E	INIT	-	$a \rightarrow a, a, c32, FRM, a(init), a(ik)$
0x10	LEG8_A	-	$a \rightarrow *a$
0x11	LEG16_A	-	$a \rightarrow *a$
0x12	LEG32_A	-	$a \rightarrow *a$
0x13	LEG64_A	-	$a \rightarrow *a$
0h14	LEG_A_REF	-	$a \rightarrow *a$
0x16	LEG8	c8	$- \rightarrow c8$
0x17	LEG16	c16	$- \rightarrow c16$
0x18	LEG32	c32	$- \rightarrow c32$
0x19	LEG64	c64	$- \rightarrow c64$
0x20	PAK8	-	$a, w8 \rightarrow -$
0x21	PAK16	-	$a, w16 \rightarrow -$
0x22	PAK32	-	$a, w32 \rightarrow -$
0x23	PAK64	-	$a, w64 \rightarrow -$
0h24	DEREF_PAK	-	$a, w64 \rightarrow -$
0x30	TCONST	a	$- \rightarrow t$
0x31	TPLAK	-	$t_1, t_2 \rightarrow t_{1,2}$
0x32	TWEG	-	$t \rightarrow -$
0x33	TLEG_A	-	$a \rightarrow t = *a$
0h34	TGLK	-	$t_1, t_2 \rightarrow c32$
0h35	TONGLK	-	$t_1, t_2 \rightarrow c32$
0h36	TPAK	-	$a, t \rightarrow -, *a = t$
0h37	TLEN	-	$t \rightarrow w32(len)$
0h38	TIDX	-	$t, w32(idx) \rightarrow w8(t[idx])$

0x40	GLK8	-	$w8, w8 \rightarrow w32$
0x41	ONGLK8	-	$w8, w8 \rightarrow w32$
0x42	GLK16	-	$w16, w16 \rightarrow w32$
0x43	ONGLK16	-	$w16, w16 \rightarrow w32$
0x44	GLK32	-	$w32, w32 \rightarrow w32$
0x45	ONGLK32	-	$w32, w32 \rightarrow w32$
0x46	GLK64	-	$w64, w64 \rightarrow w32$
0x47	ONGLK64	-	$w64, w64 \rightarrow w32$
0x48	VERGLK_N8	test	$w8, w8 \rightarrow w32$
0x49	VERGLK_N16	test	$w16, w16 \rightarrow w32$
0x4A	VERGLK_N32	test	$w32, w32 \rightarrow w32$
0x4B	VERGLK_N64	test	$w64, w64 \rightarrow w32$
0x4C	VERGLK_Z8	test	$w8, w8 \rightarrow w32$
0x4D	VERGLK_Z16	test	$w16, w16 \rightarrow w32$
0x4E	VERGLK_Z32	test	$w32, w32 \rightarrow w32$
0x4F	VERGLK_Z64	test	$w64, w64 \rightarrow w32$
0x50	VERGLK_R32	test	$w32, w32 \rightarrow w32$
0x51	VERGLK_R64	test	$w64, w64 \rightarrow w32$
0x60	PLUS_N8	-	$w8, w8 \rightarrow w8$
0x61	PLUS_N16	-	$w16, w16 \rightarrow w16$
0x62	PLUS_N32	-	$w32, w32 \rightarrow w32$
0x63	PLUS_N64	-	$w64, w64 \rightarrow w64$
0x64	PLUS_Z8	-	$w8, w8 \rightarrow w8$
0x65	PLUS_Z16	-	$w16, w16 \rightarrow w16$
0x66	PLUS_Z32	-	$w32, w32 \rightarrow w32$
0x67	PLUS_Z64	-	$w64, w64 \rightarrow w64$
0x68	MIN_N8	-	$w8, w8 \rightarrow w8$
0x69	MIN_N16	-	$w16, w16 \rightarrow w16$
0x6A	MIN_N32	-	$w32, w32 \rightarrow w32$
0x6B	MIN_N64	-	$w64, w64 \rightarrow w64$
0x6C	MIN_Z8	-	$w8, w8 \rightarrow w8$
0x6D	MIN_Z16	-	$w16, w16 \rightarrow w16$
0x6E	MIN_Z32	-	$w32, w32 \rightarrow w32$
0x6F	MIN_Z64	-	$w64, w64 \rightarrow w64$

0x70	KEER_N8	-	$w8, w8 \rightarrow w8$
0x71	KEER_N16	-	$w16, w16 \rightarrow w16$
0x72	KEER_N32	-	$w32, w32 \rightarrow w32$
0x73	KEER_N64	-	$w64, w64 \rightarrow w64$
0x74	KEER_Z8	-	$w8, w8 \rightarrow w8$
0x75	KEER_Z16	-	$w16, w16 \rightarrow w16$
0x76	KEER_Z32	-	$w32, w32 \rightarrow w32$
0x77	KEER_Z64	-	$w64, w64 \rightarrow w64$
0x78	DEEL_N8	-	$w8, w8 \rightarrow w8$
0x79	DEEL_N16	-	$w16, w16 \rightarrow w16$
0x7A	DEEL_N32	-	$w32, w32 \rightarrow w32$
0x7B	DEEL_N64	-	$w64, w64 \rightarrow w64$
0x7C	DEEL_Z8	-	$w8, w8 \rightarrow w8$
0x7D	DEEL_Z16	-	$w16, w16 \rightarrow w16$
0x7E	DEEL_Z32	-	$w32, w32 \rightarrow w32$
0x7F	DEEL_Z64	-	$w64, w64 \rightarrow w64$
0x80	VERLAAT	-	$- \rightarrow -$
0x81	ROEP	c16	$vw, FRM, a, p_0, \cdots, p_n \rightarrow vw$
0x82	GANAAR	c32	$- \rightarrow -$
0x83	vmROEP	c16,a	$- \rightarrow -$
0x90	WAAR8	c32	$w8 \rightarrow -$
0x91	ONWAAR8	c32	$w8 \rightarrow -$
0x92	WAAR16	c32	$w16 \rightarrow -$
0x93	ONWAAR16	c32	$w16 \rightarrow -$
0x94	WAAR32	c32	$w32 \rightarrow -$
0x95	ONWAAR32	c32	$w32 \rightarrow -$
0x96	WAAR64	c32	$w64 \rightarrow -$
0x97	ONWAAR64	c32	$w64 \rightarrow -$
0xA0	DEREF	-	$a \rightarrow a$
0xA1	REF	-	$a \rightarrow a$
0xA2	WEG	c8 (n)	$w8_1, \cdots, w8_n \rightarrow -$
0hA3	DEREF_A	-	$a \rightarrow a$
0hA4	NWRIJ	c8 (id), c16 (elen)	$c32 \rightarrow a$
0hA5	NWSJA	c64 (adr), c32 (len)	$- \rightarrow a$

0hA6	VERSTEL	c16 (elen)	$a, w32 \rightarrow -$
0xB0	N8alsN16	-	$w8 \rightarrow w16$
0xB1	N8alsN32	-	$w8 \rightarrow w32$
0xB2	N8alsN64	-	$w8 \rightarrow w64$
0xB3	N16alsN8	-	$w16 \rightarrow w8$
0xB4	N16alsN32	-	$w16 \rightarrow w32$
0xB5	N16alsN64	-	$w16 \rightarrow w64$
0xB6	N32alsN8	-	$w32 \rightarrow w8$
0xB7	N32alsN16	-	$w32 \rightarrow w16$
0xB8	N32alsN64	-	$w32 \rightarrow w64$
0xB9	N64alsN8	-	$w64 \rightarrow w8$
0xBA	N64alsN16	-	$w64 \rightarrow w16$
0xBB	N64alsN32	-	$w64 \rightarrow w32$
0xBC	Z8alsZ16	-	$w8 \rightarrow w16$
0xBD	Z8alsZ32	-	$w8 \rightarrow w32$
0xBE	Z8alsZ64	-	$w8 \rightarrow w64$
0xBF	Z16alsZ8	-	$w16 \rightarrow w8$
0xC0	Z16alsZ32	-	$w16 \rightarrow w32$
0xC1	Z16alsZ64	-	$w16 \rightarrow w64$
0xC2	Z32alsZ8	-	$w32 \rightarrow w8$
0xC3	Z32alsZ16	-	$w32 \rightarrow w16$
0xC4	Z32alsZ64	-	$w32 \rightarrow w64$
0xC5	Z64alsZ8	-	$w64 \rightarrow w8$
0xC6	Z64alsZ16	-	$w64 \rightarrow w16$
0xC7	Z64alsZ32	-	$w64 \rightarrow w32$
0hD0	EN8	-	$w8, w8 \rightarrow w8$
0hD1	EN16	-	$w16, w16 \rightarrow w16$
0hD2	EN32	-	$w32, w32 \rightarrow w32$
0hD3	EN64	-	$w64, w64 \rightarrow w64$
0hD4	OF8	-	$w8, w8 \rightarrow w8$
0hD5	OF16	-	$w16, w16 \rightarrow w16$
0hD6	OF32	-	$w32, w32 \rightarrow w32$
0hD7	OF64	-	$w64, w64 \rightarrow w64$
0hD8	EXLOF8	-	$w8, w8 \rightarrow w8$

0hD9	EXLOF16	-	$w16, w16 \rightarrow w16$
0hDA	EXLOF32	-	$w32, w32 \rightarrow w32$
0hDB	EXLOF64	-	$w64, w64 \rightarrow w64$
0hF0	R32alsR64	-	$w32 \rightarrow w64$