VRIJE UNIVERSITEIT BRUSSEL

Proef ingediend met het oog op het behalen
van de graad van Bachelor in de Computerwetenschappen

ONDERSTEUNING VOOR R5RS-SCHEME IN HET SCALA-AM FRAMEWORK

UITBREIDING EN BENCHMARKING VAN SCALA-AM

Jens Van der Plas
3BA Computerwetenschappen
Rolnummer 0513499
jevdplas@vub.be

Promotor: Prof. Dr. Coen De Roover
Begeleiders: Maarten Vandercammen, Quentin Stiévenart

WETENSCHAPPEN & BIO-INGENIEURSWETENSCHAPPEN
2016-2017

Inhoudsopgave

Inleiding

Dit verslag is het eindrapport van de bachelorproef. Het doel van deze bachelorproef is het uitbreiden van SCALA-AM, een framework dat gebruikt wordt voor statische codeanalyse van Scheme-programma's die aan de R5RS Scheme-standaard voldoen. Op dit moment worden nog niet alle R5RS-taalfeatures door SCALA-AM ondersteund. Dit heeft als gevolg dat Scheme-programma's die van niet-ondersteunde R5RS-features, zoals quasiquoting, macro's en sommige taalprimitieven, gebruikmaken niet met behulp van het framework geanalyseerd kunnen worden.

Deze bachelorproef bestaat uit verschillende delen. In een eerste deel werden verschillende nieuwe taalprimitieven, namelijk `sin`, `cos`, `tan`, `sqrt` en `round`, aan het framework toegevoegd, zodat nu ook programma's die van deze primitieven gebruikmaken geanalyseerd kunnen worden. In een tweede deel werd een basisimplementatie voor quasiquoting aan het framework toegevoegd, een feature die onder andere voor het construeren van lijsten gebruikt wordt. In een derde deel werden bestaande Scheme-programma's, die al dan niet reeds door SCALA-AM geanalyseerd kunnen worden, verzameld. Dit laat toe om een beeld te krijgen van welke niet-ondersteunde features het meest gebruikt worden, zodat de meest gebruikte features in de toekomst door SCALA-AM-ontwikkelaars ondersteund kunnen worden. Deze resultaten gaven aanleiding tot de uitbreiding van het eerste deel van de bachelorproef, waarin dan ook de `append`-primitieve geïmplementeerd werd.

De in deze bachelorproef geïmplementeerde features komen niet altijd overeen met de features die het meest gebruikt worden. Voor de implementatie van vele van deze meest gebruikte features is immers domeinspecifieke kennis omtrent abstracte interpretatie, de techniek die het framework gebruikt voor het analyseren van programma's, nodig. Deze techniek valt echter buiten het bestek van de thesis. In plaats daarvan werd gefocust op veelgebruikte features die deze domeinkennis niet vereisen.

Tijdens de implementatie van `append` en quasiquoting werden enkele problemen vastgesteld die ervoor zorgden dat de implementatie van deze features bemoeilijkt werd, waardoor deze slechts deels geïmplementeerd werden. Deze problemen worden dan ook uitvoerig beschreven. De volledige implementatie van deze features valt buiten het bestek van deze bachelorproef, maar we bespreken wel, met aandacht voor alle potentiële complexiteiten, een mogelijke implementatie die als leidraad voor de verdere ontwikkeling van SCALA-AM gebruikt kan worden.

Het vervolg van dit verslag is als volgt gestructureerd: in hoofdstuk 1 geven we een overzicht van de structuur en werking van SCALA-AM. In hoofdstuk 2 beschrijven we hoe de nieuwe primitieven aan het framework toegevoegd werden, waarna in hoofdstuk 3 besproken wordt hoe quasiquoting toegevoegd werd. In hoofdstuk 4 worden de resultaten van het benchmarken toegelicht. Tot slot geven we in hoofdstuk 5 een algemene conclusie.

1 SCALA-AM

Het doel van deze bachelorproef is het uitbreiden van SCALA-AM, een statisch analyseframework voor Scheme dat aan het Software Languages Lab van de Vrije Universiteit Brussel ontwikkeld werd. In deze sectie beschrijven we eerst het gebruiksdomein van het framework. Daarna wordt de algemene structuur van het framework toegelicht en wordt de werking van het framework voor de evaluatie van een Scheme-programma geïllustreerd.

SCALA-AM is een in Scala geïmplementeerd framework dat gebruikt wordt voor het uitvoeren van statische analyse op computerprogramma's. In het geval van SCALA-AM worden deze statische analyses van computerprogramma's uitgevoerd door middel van een *abstracte interpreter* die het programma op een *abstracte wijze* zal interpreteren en zodoende een statische analyse van het programma onder evaluatie zal uitvoeren; dit in tegenstelling tot een *concrete interpreter* waar een programma op een *concrete wijze* geïnterpreteerd wordt, namelijk op de manier die men bij het gewoon uitvoeren van het programma zou verwachten. SCALA-AMs abstracte interpreter werd geconstrueerd door het systematisch abstraheren van een concrete interpreter, een techniek die een zogenaamde *abstract abstract machine* (AAM) oplevert [15]. Hoewel deze methodiek en het statisch-analyseconcept buiten het kader van deze bachelorproef vallen, moet hiermee rekening gehouden worden tijdens het uitbreiden van het framework. Daarom lichten we eerst het verschil tussen een concrete en abstracte interpreter toe.

Abstracte interpretatie

Om een programma normaal uit te voeren gebruikt men een concrete interpreter. Dit soort interpreter zal de instructies van een programma stap voor stap uitvoeren. Tijdens deze interpretatie zal de interpreter datawaarden concreet voorstellen, bijvoorbeeld het getal `6.28`, de string `"concreet"` en de boolean `true`. In tegenstelling tot een concrete interpreter voert een abstracte interpreter het programma niet echt uit, maar voert het een statische analyse uit. Statische programma-analyse kan onder andere gebruikt worden voor het vinden van fouten in de code, zoals typefouten in een dynamisch getypeerde taal, of voor het genereren van codemetrieken, zoals de coverage van een testsuite.

Voor abstracte interpretatie zullen datawaarden vaak op een alternatieve manier voorgesteld worden; de abstracte interpreter maakt geen gebruik van concrete waarden, maar zal een soort *abstracte waarden* gebruiken. De vorm van deze abstracte waarden kan verschillen, afhangende van onder andere de soort analyse die men wil uitvoeren. Voorbeelden van abstracte waarden zijn datatypes zoals `Int` en `String`, die bijvoorbeeld bij typeanalyses gebruikt kunnen worden, en verzamelingen van concrete waarden, zoals `{1, 2, 3}`.

Het doel van SCALA-AM is om op een eenvoudige wijze zowel concrete als abstracte interpreters te bouwen. Hiervoor werd het framework, bestaande uit verschillende onderdelen die ieder een bepaalde interface moeten implementeren, modulair opgebouwd. Dit zorgt ervoor dat de verschillende onderdelen van het systeem een lage koppeling vertonen. Op deze manier is het mogelijk om een welbepaald onderdeel van het framework te vervangen door een gelijksoortig onderdeel om zo bijvoorbeeld een andere statische analyse uit te kunnen voeren. Deze modulariteit laat ook toe om programma's die in andere talen geschreven zijn te analyseren.

Om het framework te kunnen uitbreiden is het nodig de structuur ervan te begrijpen en te weten hoe de verschillende onderdelen van het framework samenwerken. In het vervolg van deze sectie worden daarom kort de onderdelen van het framework besproken en wordt toegelicht hoe een programma concreet geëvalueerd wordt. De hoge graad van modulariteit zorgt ervoor dat SCALA-AM een gecompliceerd framework is.

1.1 Onderdelen

In deze paragraaf worden kort enkele onderdelen van SCALA-AM toegelicht. Het framework bestaat uit vijf hoofdcomponenten en verschillende hulpcomponenten [15][17]. De hoofdcomponenten zijn:

- de semantische component van het framework die de semantiek van de geïnterpreteerde taal voorstelt;
- de lattices die mogelijk abstracte datawaarden voorstellen;
- de machinecomponent van het framework die de abstracte of concrete interpretatie van een programma leidt;
- geheugenadressen gebruikt om waarden op te slaan;

- timestamps die de abstracte interpreter de notie van tijd geven.

Adressen en timestamps zijn belangrijk voor de abstracte interpretatie van computerprogramma's, maar vallen buiten het bestek van deze bachelorproef. We zullen deze daarom enkel vernoemen waar absoluut noodzakelijk.

De koppeling tussen deze componenten dient zo laag mogelijk gehouden te worden om ervoor te zorgen dat de modulariteit van het framework zo hoog mogelijk blijft. Verschillende hulpcomponenten binnen het framework zorgen echter voor een hogere koppeling tussen de componenten. Voorbeelden van deze hulpcomponenten zijn acties, geheugenadressen en de store. Acties worden verderop in deze paragraaf toegelicht.

De store wordt gebruikt voor het opslaan van waarden en is als het ware een voorstelling van het heapgeheugen; het beeldt adressen op abstracte waarden af. Omwille van de abstracte interpretatie is het echter nodig om slechts van een eindig aantal geheugenadressen gebruikt te maken. De reden hiervoor valt echter buiten het bestek van deze bachelorproef.

In het vervolg van deze paragraaf lichten we enkele van de belangrijkste onderdelen van het framework toe. Omdat de implementatie van het framework momenteel voornamelijk toegespitst is op het gebruik van de taal Scheme, zullen voornamelijk voorbeelden uit deze taal aangehaald worden.

1.1.1 Lattices als voorstelling van datawaarden

Iedere programmeertaal kent primitieve datawaarden, zoals integers, pairs en strings. Tijdens de programma-analyse wordt een dergelijke waarde op een abstracte manier voorgesteld als een instantie van een zogenaamde lattice. Hoe deze abstracte waarde, een latticewaarde genaamd, er juist uitziet, hangt onder andere van de soort analyse die uitgevoerd wordt af. Een abstracte voorstelling van een datawaarde is bijvoorbeeld diens type, zoals `Int` of `Boolean`, of een verzameling van concrete waarden. In het eerste geval geeft de abstracte waarde aan welk datatype een datawaarde heeft, hetgeen dan bijvoorbeeld gebruikt kan worden voor het opsporen van typefouten. In het tweede geval houdt de interpreter de mogelijke waarden voor een datawaarde bij [17]. De implementatie van de lattices bepaalt welk abstract domein gebruikt wordt en hoe de data exact voorgesteld wordt. Het gekozen abstract domein bepaalt mede de nauwkeurigheid van de door het framework uitgevoerde analyse.

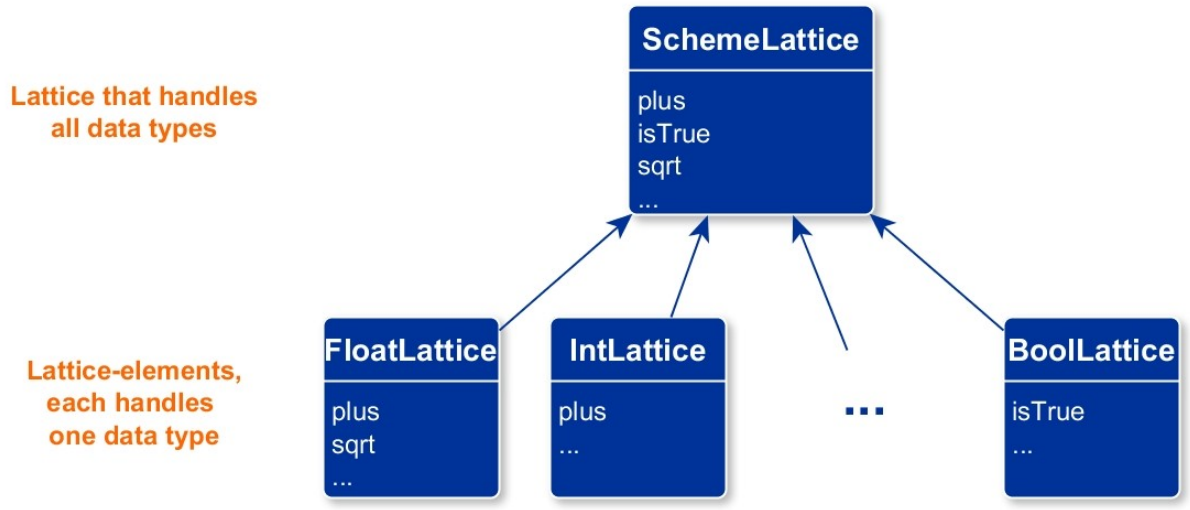
Op primitieve datawaarden moeten ook bewerkingen uitgevoerd kunnen worden. Het moet immers mogelijk zijn om twee getallen op te tellen of om te kunnen bepalen of een boolean waar of onwaar is. Aangezien de voorstelling van datawaarden van de lattice-implementatie afhankelijk is, moeten de lattices voorzien in een manier om de voorgestelde datawaarden te manipuleren. De `SchemeLattice` voorziet bijvoorbeeld een methode `plus` die twee latticewaarden optelt [17]. De door de lattices aangeboden functionaliteiten zijn geen taalprimitieven, maar zullen door de semantische component van het framework gebruikt worden voor de opbouw van de primitieven. De modulariteit van het framework laat immers toe om een bepaald lattice-type voor de implementatie van verschillende programmeertalen te gebruiken. In het vervolg van dit document zullen we de term *basisfunctionaliteit* hanteren voor de door de lattices aangeboden functionaliteiten.

Programma's kunnen ook fouten bevatten, zoals typefouten in een dynamisch getypeerde taal: men probeert bijvoorbeeld door een string te delen of een boolean als functie aan te roepen. Het doel van statische codeanalyse is het vinden van deze fouten. Bij het aantreffen van een fout legt SCALA-AM deze fout vast en documenteert het deze.

Bepalen of een operatie al dan niet correct kan verlopen is niet eenvoudig wanneer men onvolledige informatie over een bepaalde variabele heeft. Indien het type niet gekend is, hoe kan er dan nagegaan worden of een optelling zal falen? Stel bijvoorbeeld dat van een variabele `V` gekend is dat het een boolean of een string is. Dit kan in een lattice bijvoorbeeld door de verzameling `{Bool, String}` voorgesteld worden. Wat gebeurt er nu indien we de basisfunctionaliteit `string-length` op `V` aanroepen? Zoals dit voorbeeld aangeeft, is het dus mogelijk dat een operatie zowel slaagt als faalt indien niet alle informatie over een datawaarde gekend is. SCALA-AM introduceert hiervoor de `MayFail` monad die het resultaat van een dergelijke operatie – of deze nu slaagt, faalt of slaagt en faalt tegelijkertijd – op een praktische manier weer kan geven. Bij de aanroep van een basisfunctionaliteit van een lattice wordt geen nieuwe latticewaarde teruggegeven, maar een instantie van de `MayFail` mo-

nad. Informatie uit verschillende `MayFail` monads kan met behulp van een methode `bind` gecombineerd worden [14][17]. Het concept monad is een begrip uit het functioneel programmeren en valt buiten de context van deze bachelorproef.

De implementatie van de lattices is dubbellaags om de modulariteit van het framework te verhogen. Voor ieder datatype wordt er een zogenaamd lattice-element gedefinieerd dat gebruikt wordt om de datawaarden van dat bepaalde datatype voor te stellen. Zo stelt een instantie van `BoolLattice` een boolean voor en biedt dit lattice-element bepaalde basisfunctionaliteiten aan, bijvoorbeeld om na te gaan of de voorgestelde boolean waar of onwaar is. Deze lattice-elementen worden dan gecombineerd in één lattice die met de verschillende datatypes overweg kan. Voor Scheme is dit de `SchemeLattice` [17]. Het framework heeft bijvoorbeeld een `IntLattice` en een `FloatLattice` die allebei een basisfunctionaliteit `plus` aanbieden, maar enkel de `SchemeLattice` biedt een basisfunctionaliteit `plus` aan die zowel met integers als floating-pointgetallen kan werken [13]. Meestal zal de `SchemeLattice` slechts het type van de argumenten controleren om na te gaan of de operatie uitgevoerd kan worden. Indien dat het geval is, zal de aanroep naar het juiste lattice-element gedelegeerd worden. In het andere geval wordt een error teruggegeven. Figuur 1 geeft deze opbouw schematisch weer.



Figuur 1 *Schematische opbouw van de lattices. De SchemeLattice biedt de zogenaamde "basisfunctionaliteit van de lattices" aan en maakt hiervoor gebruik van de door de lattice-elementen aangeboden functionaliteit.*

De lattice-elementen kunnen op verschillende manieren geïmplementeerd worden, waardoor de datawaarden in verschillende abstracte domeinen voorgesteld kunnen worden. Momenteel bevat het framework reeds enkele implementaties voor de lattice-elementen. Voorbeelden van lattice-elementimplementaties voor integers zijn `ConcreteInteger`, waarbij een integer als een concrete waarde voorgesteld wordt, en `BoundedInteger`, waarbij de mogelijke waarden van een integer als een interval bijgehouden worden. Elke implementatie van een lattice-element dient wel steeds dezelfde functionaliteit aan te bieden. Op deze manier kan de implementatie van de lattice-elementen gewijzigd worden zonder de `SchemeLattice` aan te passen.

1.1.2 Semantiek als handleiding voor de programma-interpretatie

Een tweede belangrijk onderdeel van SCALA-AM is de semantische component van het framework. Dit onderdeel stuurt de interpretatie van een programma. De semantiek geeft aan hoe de programmatekst naar een makkelijk werkbare representatie van het programma, een *abstract syntax tree* (AST), omgezet moet worden, alsook hoe expressies die met behulp van deze representatie voorgesteld worden geëvalueerd moeten worden. Hiervoor werkt de semantische component nauw samen met de machinecomponent.

Om een bepaalde expressie te evalueren, doet de machinecomponent van het framework beroep op de door de semantiek aangeboden functionaliteit. Voor de evaluatie van een programma biedt de semantiek namelijk twee functies, `stepEval` en `stepKont`, aan. De `stepEval`-functie van de semantiek ontvangt een (sub-)expressie en gaat na hoe deze geëvalueerd moet worden. Hierop geeft de semantische component instructies, onder de vorm van acties, aan de machine die deze acties dan uitvoert. Wanneer een expressie volledig geëvalueerd werd en een waarde bereikt is, zal de machine de `stepKont`-functie van de semantiek oproepen. Deze procedure zal de

bekomen waarde gebruiken om de evaluatie verder te zetten en geeft eveneens acties terug die de machine instrueren hoe de evaluatie verdergezet moet worden [14][15][17].

De semantiek voorziet verder ook in taalspecifieke primitieve functies. Deze primitieven maken gebruik van de door de lattices aangeboden basisfunctionaliteiten om datawaarden te manipuleren. Voor de implementatie van de Scheme-primitieven wordt er een onderscheid gemaakt tussen twee soorten primitieven. De eerste soort zijn primitieven die geen informatie op bepaalde geheugenadressen lezen of naar bepaalde geheugenadressen schrijven. Voorbeelden van dergelijke primitieven zijn `+`, `round` en `not`. De tweede soort zijn primitieven die wel met geheugenadressen werken door onder andere data uit het heapgeheugen in te lezen of nieuwe waarden op de heap te alloceren. Voorbeelden van dergelijke primitieven zijn `vector`, `cons` en `list-ref`. De implementatie van primitieven van de eerste soort is vaak veel eenvoudiger dan de implementatie van primitieven van de tweede soort, aangezien bij de implementatie van de tweede soort primitieven de store gebruikt wordt.

1.1.3 Machine als leider van de interpretatie

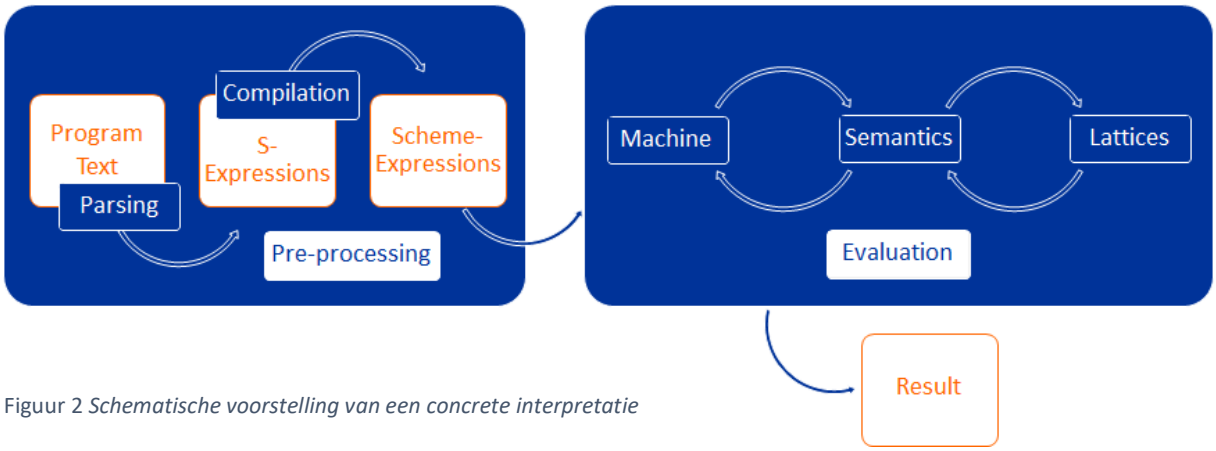
De abstracte of concrete interpretatie van een programma wordt door de machinecomponent van het framework aangestuurd. Hoewel deze component minder belangrijk is voor de uitvoering van de bachelorproef, zullen we de werking hiervan kort bespreken.

SCALA-AM maakt gebruik van een stackmachine voor de interpretatie van programma's. Dit houdt in dat de machinecomponent intern over een continuatiestack beschikt. Op deze stack worden frames die informatie voor de verdere evaluatie bevatten gepusht. De frames op de stack bepalen het verdere verloop van de evaluatie nadat de evaluatie van een bepaalde (deel-)expressie in een waarde resulteerde [17]. Omdat de evaluatiemethode voor verschillende programmeertalen verschillend is, zijn de gebruikte frames taalafhankelijk en worden ze door de semantiek gedefinieerd. Deze frames zijn dus een voorbeeld van een hulpcomponent in SCALA-AM die een hogere koppeling tussen de verschillende hoofdcomponenten veroorzaakt.

Zoals vermeld in sectie 1.1.2 werken de semantiek en de machinecomponent nauw samen. De evaluatiemethode is als volgt: de machine roept de gepaste functie van de semantiek, `stepEval` of `stepKont` op. Deze aanroep resulteert in een verzameling acties die door de machine uitgevoerd moeten worden. Voorbeelden van acties zijn het pushen of poppen van een frame op de stack of het evalueren van een bepaalde deelexpressie.

1.2 Concrete interpretatie van een Scheme-programma

In voorgaande paragraaf bespraken we kort enkele belangrijke onderdelen van het framework, alsook hun functie. Om te illustreren hoe deze onderdelen nu exact samenwerken, bespreken we in deze paragraaf hoe een concrete interpretatie van een Scheme-programma in het framework gebeurt. Figuur 2 geeft een schematisch overzicht van de stappen die tijdens de concrete interpretatie van een Scheme-programma doorlopen worden.



Figuur 2 *Schematische voorstelling van een concrete interpretatie*

1.2.1 Preprocessingsfase

De preprocessingsfase van de interpretatie wordt volledig door de semantiek uitgevoerd. Deze fase heeft als doel de uiteindelijke evaluatie mogelijk te maken. Immers, een bronbestand bestaat op zich enkel uit een opeenvolging van karakters, hetgeen niet werkbaar is aangezien strings een zeer arme datastructuur zijn. De preprocessingsfase lost dit op door het bronbestand in te lezen en in een datastructuur om te zetten, de zogenaamde abstract syntax tree. Voor de taal Scheme gebeurt dit door middel van een tweestapsproces.

In een eerste fase wordt de programmatekst in een gestructureerde expressie, een zogenaamde *s-expressie*, omgezet. Dit gebeurt met behulp van een parser die de tekst omzet en structureert. Ieder Scheme-programma bestaat immers enkel uit (geneste) s-expressies. Deze boomstructuur kan dan ook eenvoudig opgeslagen worden. De geconstrueerde s-expressie heeft dus als voordeel dat ze de programmastructuur reeds bevat. Er bestaan verschillende typen s-expressies, onder andere voor constante waarden, zoals booleans en strings, voor gequote expressies en voor identifiers.

Hoewel de s-expressies de structuur van het programma dus eenvoudig weergeven, hebben al deze expressies exact dezelfde vorm en zijn verschillende soorten expressies moeilijk te onderscheiden; het is bijvoorbeeld niet eenvoudig om een `if`-expressie van een gewone procedureaanroep te onderscheiden. Om de interpretatie te vereenvoudigen, worden deze s-expressies in een tweede fase omgezet naar zogenaamde Scheme-expressies ter constructie van een *abstract syntax tree*. Deze omzetting gebeurt door middel van compilatie. De abstract syntax tree bestaat dan uit abstracte-grammaticaelementen die verschillende soorten expressies, zoals datawaarden, een functieaanroep of een aanroep van een *Scheme special form*, voorstellen. Deze laatste categorie bestaat uit operatoren die op een speciale manier behandeld moeten worden, zoals bijvoorbeeld `if` en `define`. Op deze manier is het voor de interpreter eenvoudiger om de verschillende soorten expressies te onderscheiden dan wanneer deze met s-expressies zou werken.

Scheme kent een `quote` special form voor de constructie van literal expressions [1]. Deze literal expressies kunnen eender welke uitdrukking van Scheme voorstellen en worden niet naar een Scheme-expressie gecompileerd. Dergelijke expressies blijven s-expressies. Deze omzetting vindt plaats bij de evaluatie van de gequote expressie.

1.2.2 Evaluatiefase

Wanneer de preprocessingsfase afgelopen is, kan de verkregen Scheme-expressie geëvalueerd worden. Hoewel de machine de evaluatie aanstuurt, wordt het eigenlijke werk door de semantiek en de lattices gedaan. De werking van de machine werd reeds in vorige secties besproken.

De semantiek van Scheme beschrijft hoe een bepaalde expressie geëvalueerd moet worden. Eerst moet de operator geëvalueerd worden. Levert dit een functie op, dan worden de argumenten geëvalueerd en wordt de functieaanroep uitgevoerd. Als de operator een special form is, zal de evaluatiemethode afwijken. De gebruikte methode verschilt naargelang de special form.

Zoals vermeld maken de taalprimitieven van Scheme gebruik van de basisfunctionaliteiten van de lattices. De implementatie van de primitieven moet er wel rekening mee houden dat tijdens een abstracte interpretatie van een programma mogelijk niet alle nodige informatie voorhanden is. Wanneer men bijvoorbeeld de vierkantswortel van een getal moet nemen, kan dit enkel voor positieve getallen uitgevoerd worden, hoewel men tijdens de uitvoering van een abstracte evaluatie niet noodzakelijk weet of een getal positief of negatief is.

2 Uitbreiding van het framework met primitieven voor Scheme

Een eerste deel van deze bachelorproef houdt de toevoeging van enkele wiskundige primitieven van Scheme aan SCALA-AM in. Door de toevoeging van deze primitieven zullen meer programma's door het framework geanalyseerd kunnen worden dan voordien. Indien een programma een niet-geïmplementeerde primitieve gebruikt, zal SCALA-AM dit als een error classificeren, waardoor de programma-analyse niet verdergezet kan worden. Volgende primitieven uit de R5RS-standaard [1] werden toegevoegd: `round`, `sin`, `cos`, `tan`, `sqrt` en `append`. We lichten eerst toe hoe de transcendente functies `sin`, `cos` en `tan` toegevoegd werden alvorens `round` en `sqrt` te bespreken. Tot slot wordt de implementatie van `append` besproken.

2.1 Toevoeging van sinus, cosinus en tangens

De transcendente functies `sin`, `cos`, `tan` zijn wiskundige operatoren die op getallen opereren, hetgeen in SCALA-AM impliceert dat de implementatie van deze primitieven op de door de lattices aangeboden basisfunctionaliteit moet steunen. De lattices die instaan voor de voorstelling van datawaarden in het framework werden beschreven in sectie 1.1.1. Zoals aldaar vermeld bevat het framework verschillende implementaties voor de lattice-elementen. Het is daarom nodig om al deze implementaties uit te breiden teneinde de gewenste primitieven te implementeren.

SCALA-AM implementeert een getallenhiërarchie (number tower) die slechts uit twee niveaus bestaat, namelijk `Int` en `Float`, en daarmee van de R5RS-standaard afwijkt. Aangezien de te implementeren primitieven op beide types toegepast kunnen worden, moeten zowel `IntLattice` als `FloatLattice` basisfunctionaliteit voor deze operaties aanbieden. Voor de implementatie wordt voor een alternatieve strategie geopteerd: in plaats van beide lattice-elementen uit te breiden, wordt een integer eerst omgezet naar een floating-pointgetal alvorens de basisfunctionaliteit van een lattice-element aan te roepen. Op deze manier moet enkel `FloatLattice` uitgebreid worden. Deze strategie zorgt er dan ook voor dat `IntLattice` ongewijzigd blijft, maar het nadeel van deze strategie is dat het resultaat van een dergelijke aanroep steeds een floating-pointgetal is. Naast de getallenhiërarchie maakt de R5RS-standaard immers ook onderscheid tussen *exacte* getallen en *inexacte* getallen [1]. Aangezien deze indeling nog niet in het framework geïmplementeerd werd, is het geen probleem dat de nieuw toegevoegde basisfunctionaliteiten steeds een `Float` teruggeven.

Om `FloatLattice` uit te breiden, moeten verschillende soorten abstracte waarden, oftewel abstracte domeinen, uitgebreid worden. Een eerste domein, `Concrete` genaamd, stelt waarden als verzamelingen van concrete `Float`s van Scala voor. Het toevoegen van de transcendente functies aan dit domein werkt als volgt: pas de onderliggende functie van Scala toe op ieder element van de verzameling. Het resultaat is dan een nieuwe verzameling bestaande uit de getransformeerde waarden. Voor sinus, cosinus en tangens ziet deze definitie er als volgt uit [16]:

```
def sin(n: F): F = n.map(n => scala.math.sin(n.toDouble).toFloat)
def cos(n: F): F = n.map(n => scala.math.cos(n.toDouble).toFloat)
def tan(n: F): F = n.map(n => scala.math.tan(n.toDouble).toFloat)
```

Een tweede abstract domein, `Type` genaamd, houdt voor iedere waarde het type bij. Ieder lattice-element stelt de concrete waarde als een speciale waarde `Top` voor. Een boolean wordt dan als `Bool(Top)` voorgesteld. Naast `Top` bestaat er ook nog een speciale waarde `Bottom`. Omdat het resultaat van een aanroep van de wiskundige primitieven sinus, cosinus en tangens die aan de FloatLattice toegevoegd worden eveneens een `float` is, is het in dit domein voldoende om het argument, dat `Top` of `Bottom` zal zijn, terug te geven. Het type dat uit de aanroep van de primitieven resulteert is immers hetzelfde dan het type van het argument. In dit domein worden sinus, cosinus en tangens dus als volgt geïmplementeerd [16]:

```
def sin(n: T): T = n
def cos(n: T): T = n
def tan(n: T): T = n
```

Een derde en laatste abstract domein dat uitgebreid moest worden, `ConstantPropagation` genaamd, heeft zowel kenmerken van het `Concrete` domein als van het `Type` domein. Indien mogelijk wordt in dit domein een constante waarde bijgehouden, zoals in `Concrete`. Wanneer dit niet mogelijk blijkt, wordt op een speciale

`Top` waarde overgegaan, zoals in `Type`. De implementatie van de primitieven in dit domein is eveneens eenvoudig en zullen we daarom niet bespreken.

Een tweede stap in het toevoegen van deze primitieven is het uitbreiden van de `SchemeLattice`. De `SchemeLattice` kan met alle datatypes omgaan, maar de transcendente primitieven kunnen echter enkel op numerieke waarden toegepast worden. De `SchemeLattice` moet dus het datatype van het argument controleren en geeft een error terug wanneer dit datatype afwijkt. Het is ook de `SchemeLattice` die integers omzet naar floating-pointgetallen, zodat de functionaliteit van de `FloatLattice` gebruikt kan worden.

2.2 Toevoeging van round en sqrt

De implementatie van de primitieven `round` en `sqrt` is grotendeels gelijkaardig aan de implementatie die in sectie 2.1 besproken werd. Ook voor deze primitieven wordt enkel de `FloatLattice` uitgebreid. Er zijn echter enkele belangrijke verschillen die de implementatie compliceren.

2.2.1 Round

Scheme's `round`-primitieve verschilt van de wiskundige manier van afronden en dus ook van Scala's `round`. In Scheme wordt een waarde die zich midden tussen twee gehele getallen bevindt naar een even getal afgerond [1]. Zo wordt `3.5` naar `4` afgerond en wordt `-1.5` naar `-2` afgerond. Het verschil is subtiel maar houdt in dat het niet mogelijk is om in het domein `Concrete` de onderliggende functie van Scala te gebruiken. In plaats daarvan wordt een nieuwe methode, die een waarde op dezelfde manier als de Scheme-primitieve `round` afrondt, geïmplementeerd. Dit gebeurt met behulp van volgende methode die gebruik maakt van Scala's `round`-primitieve [16]:

```
/** Round in Scheme and Scala are different. This implements the same behaviour as
Scheme's round. */
def round(n: Float): Float = {
  val frac = n % 1 /* Fractional part of n */
  /* In the case of a fraction part equaling 0.5, rounding is done towards the even
number. */
  if ((scala.math.abs(frac) == 0.5) && (((n > 0) && ((scala.math.abs(n - frac) % 2)
== 0)) || ((n < 0) && (((n - frac) % 2) == -1)))) {
    scala.math.round(n) - 1
  }
  else {
    scala.math.round(n)
  }
}
```

In `Concrete` wordt dan van deze methode gebruikgemaakt in plaats van Scala's primitieve te gebruiken. Dit zorgt ervoor dat de Scheme-primitieve zoals verwacht werkt. De implementatie van `round` in de domeinen `Type` en `ConstantPropagation` is gelijkaardig aan die van de primitieven die voorgaand besproken werden.

2.2.2 Sqrt

Ook het implementeren van de `sqrt`-primitieve is niet triviaal. Het probleem hierbij wordt veroorzaakt door de wiskundige definitie van de vierkantswortel, die enkel voor positieve getallen gedefinieerd is. Wanneer de `sqrt`-primitieve op een negatieve waarde toegepast wordt, kan de operatie niet uitgevoerd worden en dient een error teruggegeven te worden.

Zoals beschreven in sectie 1 werkt de abstracte interpreter niet steeds met concrete waarden of is niet alle informatie over een waarde bekend. In het framework is het dan ook mogelijk dat een waarde zowel groter als kleiner dan nul tegelijkertijd is, of geen van beide. Hierdoor is het mogelijk is dat de `sqrt`-operatie faalt en slaagt tegelijkertijd. Een voorbeeld hiervan is een integer die in het `Type` domein voorgesteld wordt. Van deze integer is dan niet geweten of hij positief of negatief is, waardoor hij als zowel positief als negatief beschouwd wordt. In dit geval dienen zowel het resultaat als een error teruggegeven worden. Dit gebeurt met behulp van de `MayFail` monad, zoals beschreven in sectie 1.1.1. Hiervoor dient de `SchemeLattice` de numerieke waarde van het getal te controleren vooraleer het de functionaliteit van de lattice-elementen kan gebruiken.

2.3 Toevoeging van append

Een laatste primitieve die aan het framework toegevoegd werd is `append`. Zoals aangegeven in sectie 1.1.2 bestaan er twee typen primitieven. De primitieven `sin`, `cos`, `tan`, `round` en `sqrt` behoren tot het eerste type, namelijk tot de primitieven die geen geheugenadressen gebruiken. `Append` behoort tot het tweede type, namelijk tot de primitieven die wel geheugenadressen gebruiken. In SCALA-AM impliceert het werken met geheugenadressen het gebruik van de store, hetgeen de implementatie van een primitieve van dit type complexer maakt.

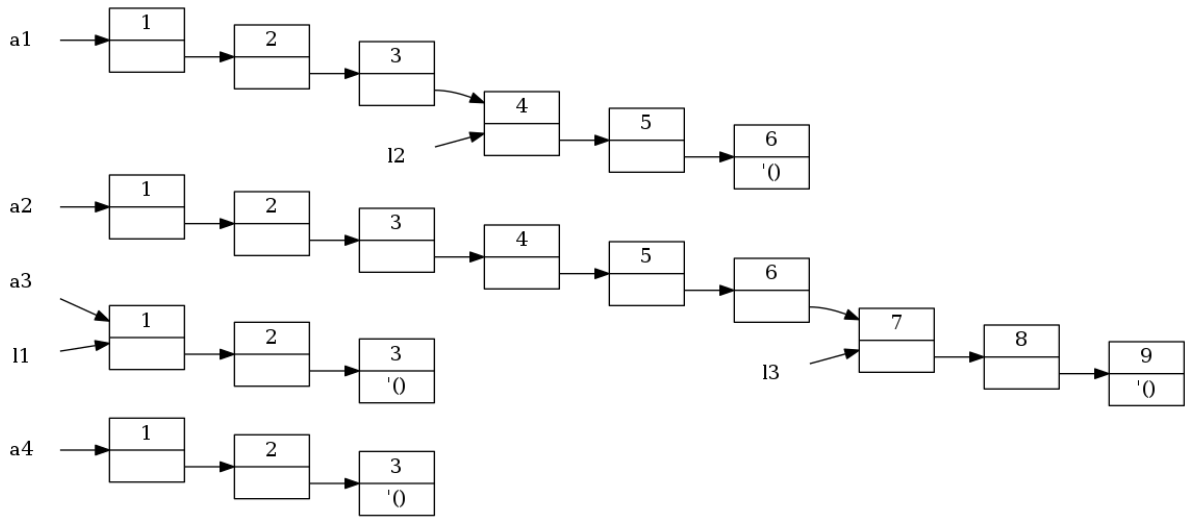
De `append`-procedure van Scheme neemt één of meerdere Scheme-lijsten als argument en construeert een nieuwe lijst die de elementen van alle argumentlijsten bevat. De Scheme-standaard geeft aan dat de nieuwe lijst zich in nieuw gealloceerd geheugen moet bevinden, maar maakt een uitzondering voor het laatste argument [1]:

> *The resulting list is always newly allocated, except that it shares structure with the last list argument. The last argument may actually be any object; an improper list results if the last argument is not a proper list.*

Om dit te verduidelijken, geven we een voorbeeld. Veronderstel dat volgend programma uitgevoerd wordt:

```
(define l1 '(1 2 3))
(define l2 '(4 5 6))
(define l3 '(7 8 9))
(define a1 (append l1 l2))
(define a2 (append l1 l2 l3))
(define a3 (append l1))
(define a4 (append l1 '()))
```

Figuur 3 geeft met behulp van een box-and-pointerdiagram aan hoe deze lijsten dan in het geheugen voorgesteld worden. Zoals op de figuur te zien is, delen zowel `a1` en `l2` als `a2` en `l3` bepaalde geheugenlocaties ten gevolge van de R5RS-specificatie. Deze specificatie zorgt er dan ook voor dat `l1` en `a3` dezelfde lijst zijn, terwijl `a4` een kopie van `l1` is en dus geen enkele cel gemeenschappelijk heeft.



Figuur 3 *Box-and-pointerdiagram*

Voor de implementatie van `append` zal dus de store gebruikt moeten worden voor het alloceren van nieuw geheugen. `Append` is de eerste primitieve die aan het framework toegevoegd wordt die zelf grote hoeveelheden geheugen moet alloceren. Dit zal voor bijkomende moeilijkheden in de implementatie, die in de volgende sectie besproken worden, zorgen. De implementatie van `append` is op dit moment enkel afgestemd op het gebruik met twee argumenten, maar dit kan eenvoudig uitgebreid worden naar een willekeurig aantal argumenten. De implementatie werkt ook enkel indien SCALA-AM als een concrete interpreter gebruikt wordt. De technische redenen hiervoor worden eveneens in volgende sectie toegelicht. Aangezien abstracte interpretatie buiten de reik-

wijdte van deze bachelorproef valt, is het voor deze bachelorproef dan ook voldoende dat de concrete interpretatie correct werkt. Toch wordt reeds zo veel mogelijk rekening gehouden met uitvoering door een abstracte interpreter.

2.3.1 Implementatie

Voor de implementatie van `append` kan als basis uitgegaan worden van een implementatie in Scheme zelf:

```
(define (append a b)
  (if (null? a)
      b
      (cons (car a)
            (append (cdr a)
                    b))))
```

Hoewel deze definitie recursief is, kan deze toch als basis voor de implementatie in SCALA-AM dienen. In de implementatie van `append` in SCALA-AM, is er echter ook nog een bijkomende test nodig om na te gaan of de eerste argumentlijst wel een correcte lijst is. In het bovenstaand voorbeeld gebeurt dit door de onderliggende Scheme-implementatie.

Bij uitvoering in een abstracte interpreter is niet steeds alle informatie voorhanden. Net zoals een operatie op een lattice kan slagen, falen of slagen en falen tegelijkertijd, is dit ook voor testen geldig. Zo kan de `null?`-test in bovenstaande code zowel `true`, `false`, `true` en `false` of noch `true` en noch `false` tegelijkertijd zijn, net zoals bij de `sqrt`-primitieve niet altijd eenduidig te bepalen is of een waarde positief of negatief is. Dit moet dan ook in de implementatie van `append` opgevangen worden. We kunnen dan ook de benodigde codestructuur met behulp van volgende pseudocode weergeven:

```
1   def append(list1, list2) {
2     nulltest := check(list1 is empty list) # Verify whether list1 is empty.
3     t := {
4       if (nulltest)
5         list2
6       else bottom # Neutral element for combine method.
7     }
8     f := {
9       if (!nulltest) {
10        constest := check(list1 is a pair) # Verify whether list1 is a pair.
11        ft := {
12          if (constest)
13            cons(head of list1, append(tail of list1, list2))
14          else bottom
15        }
16        ff := {
17          if (!constest)
18            error(Type error) # Return an abstract error representation.
19          else bottom
20        }
21        combine(ft, ff) # We need a way of combining both results.
22      }
23      else bottom
24    }
25    return combine(t, f)
```

Deze pseudocode geeft enkel de grote lijnen van de structuur weer. Zo is het bijvoorbeeld ook nodig om `append` met behulp van een fixpuntiteratie te implementeren om te garanderen dat het aanroepen van de `append`-primitieve steeds een resultaat teruggeeft, zelfs als deze aangeroepen wordt op bijvoorbeeld circulaire, en dus oneindig lange, lijsten. Wel is duidelijk te zien dat de structuur van de pseudocode overeenkomt met de structuur

van bovenstaande Scheme-code. Aan de hand van deze pseudocode wordt nu de echte implementatie besproken.

Net zoals alle waarden in SCALA-AM, worden `list1` en `list2` door de lattices voorgesteld en zijn het dan ook mogelijkerwijs abstracte waarden. Om te testen of `list1` `null` of een pair is, moet er dan ook van de door de lattices aangeboden basisfunctionaliteit gebruikgemaakt worden. Zoals aangegeven in sectie 1.1.1 is het resultaat van een dergelijke aanroep een `MayFail`. Via de `bind`-methode kan dit resultaat dan gebruikt worden om na te gaan of de test slaagde en/of/noch faalde. Het combineren van resultaten, zoals bijvoorbeeld in lijnen 21 en 25 gebeurt, houdt dan het combineren van informatie in twee `MayFail`s in. Hiervoor bestaat een methode, toevallig ook `append` genaamd, die dan ook gebruikt wordt. Deze `append` heeft echter niets te maken met de Scheme-primitieve waarvan de implementatie in deze paragraaf besproken wordt.

Gebruik van geheugenadressen

Het moeilijkste onderdeel van deze implementatie is het gebruik van de store en de bijhorende geheugenadressen. Om een geheugenadres voor een pair te alloceren, wordt onder andere de timestamp van het framework gebruikt, evenals een expressie die aan het geheugenadres van de cel gekoppeld moet worden. Een eerste probleem hierbij wordt veroorzaakt door het feit dat de nodige expressie niet beschikbaar is.

De `append`-primitieve moet echter een groot aantal geheugenadressen alloceren. Omwille van een technische reden die met abstracte interpretatie te maken heeft, zou de timestamp idealiter na iedere geheugenallocatie verhoogd moeten worden. Dit is in de huidige implementatie van SCALA-AM niet mogelijk, aangezien deze implementatie ervan uitgaat dat enkel de machine te timestamp verhoogt. Daarom is er in de huidige implementatie geen manier om de verhoogde timestamp aan de machine terug te geven. Dit is problematisch, aangezien de machine de nieuwe timestamp nodig heeft vooraleer de volgende iteratie aan te vatten. Zowel het beschikbaar stellen van de expressie die nodig is voor de allocatie van een geheugenadres voor een pair als het mogelijk maken voor de semantiek om een gewijzigde timestamp aan de machine terug te geven, zouden enorm veel werk vragen. Immers, hiervoor moeten verschillende onderdelen van het framework gewijzigd worden, alsook de interface van de semantiek. De hoeveelheid werk dat hiervoor vereist zou zijn overstijgt wat in deze bachelorproef mogelijk is.

Om deze problemen te omzeilen, werd er voor de uitvoering van deze bachelorproef een nieuw type geheugenadres gedefinieerd. In plaats van de globale timestamp te gebruiken voor de aanmaak van een geheugenadres, wordt er voor dit type een speciale counter bijgehouden. Bij de aanmaak van een adres, wordt de counter verhoogd, zodat ieder adres uniek is. Deze counter vereist echter wel een muteerbare variabele, terwijl SCALA-AM in functionele stijl geschreven werd. Door er ook voor te zorgen dat voor de aanmaak van dit nieuwe adrestype geen expressie vereist is, werden beide problemen die met deze geheugenadressen samenhangen opgelost. Deze oplossing zorgt er echter wel voor dat de `append`-primitieve van Scheme momenteel enkel correct werkt wanneer SCALA-AM als een concrete interpreter gebruikt wordt.

Uitbreiding van de store

De store bindt waarden aan geheugenadressen. Bij iedere nieuwe binding dient de store dan ook uitgebreid te worden. Bovendien, aangezien `append` recursief geïmplementeerd is, dient de uitgebreide store telkens mee als resultaat teruggegeven te worden, zodat deze bij het terugkomen uit de recursie verder uitgebreid kan worden. Het resultaat van iedere recursieve aanroep is een `MayFail`. Iedere `MayFail` is in staat om meerdere waarden samen bij te houden. Daarom werd getracht om ook de uitgebreide store als onderdeel van de `MayFail` als resultaat terug te geven.

Om dit te kunnen doen, moet de store als een zogenaamde *monoid* gedefinieerd worden. Het begrip monoid vindt zijn oorsprong in de wereld van het functioneel programmeren. Het is belangrijk te weten dat een monoid, net zoals `MayFail`, een methode `append` aanbiedt, waarmee de informatie uit twee monoids samengevoegd kan worden. Deze `append` blijkt er echter voor te zorgen dat de uitbreidingen van de store niet correct gebeuren, waardoor deze oplossing niet bruikbaar is.

Ook dit probleem valt te omzeilen door van een muteerbare variabele gebruik te maken. Deze variabele kan dan telkens gebruikt worden om de nieuwe, uitgebreide store in op te slaan. Na terugkeer uit alle recursieve aanroepen van de geïmplementeerde `append`, kan deze uitgebreide store dan mee teruggegeven worden als resultaat van de aanroep van de functie.

2.4 Conclusie

In deze sectie werd beschreven hoe verschillende primitieven van Scheme aan SCALA-AM toegevoegd werden. Om de primitieven `sin`, `cos`, `tan`, `round` en `sqrt` toe te voegen, moest de basisfunctionaliteit van de lattices uitgebreid worden. Hiervoor moesten verschillende abstracte domeinen, zoals `Concrete` en `Type`, uitgebreid worden. Omdat de functionaliteit van de `round`-primitieve van Scala verschilt van deze van Scheme, moet er voor `Concrete` een andere implementatie gebruikt worden. Voor de `sqrt`-primitieve moest er dan weer speciale aandacht besteed worden aan het argument, aangezien niet steeds eenduidig bepaald kan worden of dat positief of negatief is.

De implementatie van al deze primitieven is relatief eenvoudig, aangezien ze de store niet gebruiken. De implementatie van `append` daarentegen is complexer. Een eerste moeilijkheid is dat de gangbare geheugenadressen voor pairs niet eenvoudig voor deze primitieve te gebruiken zijn, aangezien `append` zelf de timestamp moet aanpassen. Een tweede moeilijkheid was het gevolg van de uitbreiding van de store, waarvan de uitgebreide versie mee als het resultaat van een aanroep teruggegeven diende te worden. Beide problemen konden opgelost worden door het gebruik van een muteerbare variabele. Het gebruik van de timestamp kon immers omzeild worden door een nieuw soort geheugenadressen, dat van een muteerbare variabele gebruikmaakt, te definiëren.

3 Implementatie van quasiquoting

Een tweede deel van de bachelorproef heeft als doel om quasiquoting te implementeren. Quasiquoting is een Scheme-feature die gebruikt wordt voor het construeren van lijsten en vectoren waarvan de structuur slechts gedeeltelijk op voorhand gekend is. Deze feature voorziet in drie special forms, namelijk `quasiquote`, `unquote` en `unquote-splicing` die respectievelijk ook met de backquote (`` ` ``), komma (`,`) en komma-apenstaart (`,@`) genoteerd kunnen worden [1]. We geven enkele voorbeelden van het gebruik van deze feature uit [17]:

```
; Een komma zorgt ervoor dat een bepaald argument geëvalueerd wordt.
`(cons (+ 1 2) ,(+ 1 2))
> (cons (+ 1 2) 3)
`#(1 2 ,(+ 1 2) ,(car (list 4)))
> #(1 2 3 4)
; Een komma gevolgd door een apenstaart haalt de haken van een lijst weg.
; Hierdoor wordt de lijst uitgevlakt.
`((list 1 2 3) ,(list 1 2 3) ,@(list 1 2 3))
> ((list 1 2 3) (1 2 3) 1 2 3)
```

Eenvoudige vormen van quasiquoting laten toe om de expressies te herschrijven. De expressies in voorgaand voorbeeld kunnen dan als volgt geschreven worden, zodat de semantiek van de expressies bewaard blijft:

```
(list `cons `(+ 1 2) (+ 1 2))
> (cons (+ 1 2) 3)
(vector `1 `2 (+ 1 2) (car (list 4)))
> #(1 2 3 4)
(list `(list 1 2 3) (list 1 2 3) 1 2 3)
> ((list 1 2 3) (1 2 3) 1 2 3)
```

Het derde voorbeeld is echter enkel eenvoudig om te zetten omdat de te splicen lijst eenvoudig af te leiden is uit het argument van de splicingprimitieve en de exacte argumenten van de quasiquote-primitieve dus gekend zijn. In het algemeen is dit niet het geval, aangezien het argument van de splicingprimitieve naar eender welke lijst kan evalueren. Een voorbeeld is de expressie `(quasiquote (0 ,@(a-function 1 2) 3))`, waarbij het niet duidelijk is naar welke lijst `(a-function 1 2)` evalueert. Deze omzetting impliceert echter al een eerste mogelijke implementatiestrategie voor quasiquoting, namelijk het omzetten van expressies die quasiquoting gebruiken naar equivalente expressies die `list`, `vector` en `quote` gebruiken. Dit zou dan betekenen dat de parser en compiler van het framework aangepast moeten worden.

In Scheme is het echter toegelaten om primitieven te herdefiniëren. Het gebruik van bovenstaande implementatiestrategie betekent dat wanneer een van bovenstaande primitieven (`list`, `vector` of `quote`) geherdefinieerd wordt, de primitieven die quasiquoting implementeren niet meer naar behoren zouden werken. Stel bijvoorbeeld dat de `list`-primitieve als volgt geherdefinieerd werd:

```
(define (list . elms)
  (for-each (lambda (e)(display e))
            elms))
```

Indien quasiquoting dan met behulp van omzettingen geïmplementeerd zou zijn, zouden bovenstaande voorbeelden foutief geëvalueerd worden. Er wordt een expressie op het scherm geprint en het resultaat is `void`:

```
(define lijst (list `(list 1 2 3) (list 1 2 3) 1 2 3))
123(list 1 2 3)#<void>123
lijst
> #<void>
```

Het is bovendien ook niet meteen duidelijk hoe alle `quasiquote`-expressies omgezet kunnen worden, aangezien bij het nesten van quasiqote-expressies deze omzetting mogelijk een aanzienlijke complexiteit zou verkrijgen. Om deze problemen te vermijden wordt deze implementatiestrategie vermeden; in plaats daarvan zal quasiquoting als een special form in de interpreter ingebouwd worden.

In het vervolg van deze paragraaf wordt beschreven hoe quasiquoting deels geïmplementeerd werd. Deze implementatie kan echter niet alle mogelijke quasiquote-expressies afhandelen, dus worden ook de beperkingen van de huidige implementatie besproken, alsook de problemen die bij de vervollediging van de ondersteuning voor quasiquoting verwacht worden. Momenteel kunnen geneste quasiquote-expressies en geneste unquote-expressies nog niet geëvalueerd worden. Ook unquote-splicing wordt nog niet ondersteund.

3.1 Preprocessing

Iedere expressie die geëvalueerd moet worden, wordt, zoals aangegeven in sectie 1.2.1, eerst naar een s-expressie geparset. De extra symbolen, zoals de backquote, die voor quasiquoting gebruikt kunnen worden dienen dan ook door de parser ondersteund te worden. Ook de compiler, die de s-expressies daarna omzet naar Scheme-expressies, dient uitgebreid te worden, aangezien er voor quasiquoting nieuwe abstracte-grammaticaelementen nodig zijn. In deze paragraaf wordt daarom de uitbreiding van parser en compiler besproken.

Voor de uitbreiding van de compiler dienen eerst nieuwe abstracte-grammaticaelementen toegevoegd te worden. Voor unquoting en unquote-splicing wordt er telkens één nieuw element toegevoegd. Voor quasiquoting daarentegen worden er twee grammaticaelementen toegevoegd om het onderscheid te maken tussen een gequasiquote lijst, zoals `` `(1 2 3 ,(+ 4 5)) ``, en een gequasiquoot element, zoals `` `4 ``. In het eerste geval moet de evaluatie immers leiden tot de constructie van een nieuwe lijst, terwijl het tweede voorbeeld naar `4` moet evalueren en er dus geen nieuwe lijst geconstrueerd dient te worden. Het gebruik van twee abstracte-grammaticaelementen staat toe om deze twee situaties tijdens de evaluatie van een programma te onderscheiden. Er moet bijvoorbeeld een onderscheid gemaakt kunnen worden tussen een gequasiquoot element dat geen lijst is en een gequasiquote lijst van lengte één.

Tijdens de evaluatie van een quasiquote-expressie mogen niet alle deelexpressies geëvalueerd worden; enkel de expressies die geünquoot zijn mogen geëvalueerd worden. Het is echter omslachtig om tijdens de evaluatiefase van de (abstracte) interpretatie te controleren welke expressies al dan niet geëvalueerd mogen worden. Om dit te vereenvoudigen werd geopteerd om de quasiquote-expressies in de compiler deels om te vormen door de niet-geünquote expressies “handmatig” te quoten. Volgend voorbeeld geeft een dergelijke transformatie concreet weer:

```
(quasiquote (1 2 (+ 1 2) ,(+ 2 2)))  ==>  (quasiquote ('1 '2 '(+ 1 2) ,(+ 2 2)))
```

In bovenstaande expressie worden drie expressies gequoted. Het voorbeeld geeft aan dat er een expressie ontstaat die dezelfde semantische betekenis heeft dan de originele expressie, maar die makkelijker te evalueren valt. Omdat dit reeds in de compilatiefase gebeurt, wordt deze transformatie uitgevoerd op de interne representatie van de expressie en niet op de expressie zelf, zoals in het voorbeeld.

3.1.1 Omgaan met meerdere notaties voor (quasi-)quoting

De dubbele notatie voor `quasiquote`, `unquote` en `unquote-splicing`, die respectievelijk ook met backquote (`` ` ``), komma (`,`) en komma-apenstaart (`,@`) genoteerd kunnen worden, zorgt voor bijkomende complexiteit in de compiler. De verkorte notaties kunnen direct door de parser herkend worden en worden gemarkeerd als een speciaal soort s-expressie; ook constante waarden en gequote expressies worden zo gemarkeerd. Wanneer een programmeur van de primitieven gebruikmaakt, worden deze gewoon als een normale s-expressie geparset. De parser houdt geen rekening met de betekenis van de s-expressie, maar kijkt enkel naar de vorm van de programmatekst. Dit zorgt er dus voor dat `(quasiquote x)` en `` `x `` beide op een andere manier geparset worden, waardoor de compiler met twee parsevormen zou moeten kunnen omgaan.

Het is duidelijk dat de vorm die uit `` `x `` resulteert meer informatief is voor de compiler aangezien deze reeds als speciale s-expressie gemarkeerd werd. De eerste vorm zorgt bovendien voor extra moeilijkheden wanneer deze genest wordt gebruikt. Om dit te vermijden en ervoor te zorgen dat de compiler enkel met de tweede vorm rekening moet houden, wordt een extra tussenstap in de preprocessingsfase geïntroduceerd waarin er gezorgd wordt dat de eerste vorm naar de tweede vorm omgezet wordt. Dit gebeurt ook voor `unquote`, `unquote-splicing` en `quote`.

3.1.2 Verificatie van de expressie

Door de mogelijkheid om gequasiquote expressies te nesten is het moeilijker om na te gaan of de programmatekst van de gebruiker aan de R5RS-standaard voldoet. De primitieven `unquote` en `unquote-splicing` mogen immers enkel binnen een quasiquote gebruikt worden. Indien dat niet het geval is, ontstaat een ongeldige expressie na de evaluatie van een gequasiquote expressie. Volgend voorbeeld toont dit aan:

```
(define x '6)
; Dit is correct.
`(1 2 `(3 4 ,(+ 5 ,x)))
> (1 2 `(3 4 ,(+ 5 6)))
; Dit is niet correct omdat de expressie ,x niet geëvalueerd kan worden in Scheme.
`(1 2 3 4 ,(+ 5 ,x))
> ERROR - unquote: not in quasiquote in: (unquote x)
```

Bovenstaand voorbeeld toont aan dat een ongeldige Scheme-expressie resulteert indien er een bepaalde positie in de expressie is die een dieper in unquotes en unquote-splicings genest is dan in quasiquotes. In bovenstaand voorbeeld is `x` één `quasiquote`-expressie en in twee `unquote`-expressies genest.

Om dit op te sporen wordt na de compilatie gecontroleerd of bovenstaande situatie zich voordoet. Indien dat het geval is, wordt een error teruggegeven aan de gebruiker. Het programma is dan immers ongeldig en kan onmogelijk correct geïnterpreteerd worden.

Deze verificatie kan in principe ook voor de compilatie plaatsvinden. In dat geval moet een boom van s-expressies gecontroleerd worden in plaats van een AST. Het resultaat van beide opties is hetzelfde. Het is echter cruciaal om de verificatie na de transformatie die in sectie 3.1.1 beschreven werd uit te voeren. In het andere geval moet de verificatieprocedure op beide parsevormen controleren, wat de implementatie nodeloos complex zou maken.

3.2 Evaluatie

In sectie 3.1 werd reeds beschreven dat gequasiquote expressies naar twee verschillende types abstracte-grammaticaelementen geparset worden. Het eerste type omvat de quasiquotes waarvan het argument geen lijst is. Voorbeelden zijn `` `x `` en `` `,(+ 1 2) `` die respectievelijk naar `'x` en `3` evalueren. Het tweede voorbeeld is hier vrij subtiel: het lijkt namelijk dat het argument van `quasiquote` een lijst is, maar dit is niet correct; het argument van de `quasiquote` is namelijk een `unquote`-expressie. Het tweede type omvat quasiquotes waarvan het argument wel een lijst is, zoals `` `(1 2 3) `` en `` `(1 2 3 ,@(list 4 5)) `` die respectievelijk naar de lijsten `'(1 2 3)` en `'(1 2 3 4 5)` evalueren. Het onderscheid dat tussen de twee abstracte-grammaticaelementen voor lijsten gemaakt wordt is dus noodzakelijk omdat beide types een verschillende evaluatie vereisen. De evaluatie van het eerste type is redelijk eenvoudig; de evaluatie van het tweede type is complexer aangezien er een nieuwe lijst geconstrueerd dient te worden. In volgende paragrafen wordt de evaluatiemethode voor beide types besproken.

Momenteel worden slechts niet-geneste vormen van quasiquoting en unquoting ondersteund. Splicing wordt in geen enkele vorm ondersteund. In sectie 3.3 wordt uitgelegd waarom deze vormen nog niet ondersteund worden en hoe de ondersteuning hiervan aan het framework toegevoegd kan worden. Quasiquoting wordt bovendien enkel voor lijsten ondersteund, aangezien de literal notation voor vectoren nog niet door het framework ondersteund wordt.

3.2.1 Quasiquotes waarvan het argument geen lijst is

De evaluatie van Scheme-expressies gebeurt met behulp van de `stepEval`-functie van de semantiek. Het evalueren van een gequasiquote expressie waarvan het argument geen lijst is, is zeer eenvoudig: het argument kan gewoon geëvalueerd worden. Om dit te verduidelijken, bespreken we de verschillende mogelijke situaties en tonen we telkens aan dat deze methode correct is.

Een quasiquote waarvan het argument geen unquote(-splicing)-expressie is

Deze situatie heeft betrekking op expressies zoals `` `x ``. Dergelijke expressies hebben dezelfde semantiek als expressies die de `quote`-primitieve gebruiken. De expressie is dus equivalent aan `'x` [1].

Zoals aangehaald in sectie 3.1 worden expressies binnen een quasiquote die niet geünquoot zijn automatisch door de compiler gequoot. Net zoals “normale” gequote expressies, wordt er hierbij voor gezorgd dat de gequote uitdrukking niet naar een Scheme-Expressie omgezet wordt, maar een s-expressie blijft. De expressies zijn dan als het ware dubbel gequoot, namelijk `(quasiquote x)` wordt `(quasiquote 'x)`. Wanneer de semantiek die een dergelijke gequasiquote expressie moet evalueren dit doet door de expressie te evalueren, is het resultaat zoals verwacht.

Een quasiquote waarvan het argument een unquote-expressie is

Deze situatie heeft betrekking op expressies zoals `` `,x `` en `` `,(+ 1 2) ``, waarbij een expressie gequoot en meteen terug geünquoot wordt. De expressie die gequasiquoot werd, is hier dus steeds een `unquote`-expressie. Ongeacht de onzinnigheid van deze constructie kan deze in een programma voorkomen en moet deze dus correct werken. In het framework wordt een geünquote expressie momenteel gewoon geëvalueerd. Aangezien `quasiquote` en `unquote` elkaar moeten opheffen, heeft het evalueren van de `unquote`-expressie het gewenste effect.

Een quasiquote waarvan het argument een unquote-splicingexpressie is

De R5RS-standaard verklaart expressies zoals `` `,@x `` ongeldig en stelt dat `unquote-splicing` enkel in een quasiquote waarvan het argument een lijst is mag voorkomen. SCALA-AM is momenteel echter nog niet in staat om dergelijke ongeldige situaties te herkennen.

In deze sectie werd aangetoond hoe een gequasiquote expressie waarvan het argument geen lijst is geëvalueerd kan worden door het argument te evalueren. In volgende sectie wordt besproken hoe de evaluatie van een gequasiquote lijstexpressie dient te gebeuren.

3.2.2 Quasiquotes waarvan het argument een lijst is

Een quasiquote waarvan het argument een lijst is, evalueert naar een nieuwe lijst; dit in tegenstelling tot de quasiquotes die in voorgaande sectie besproken werden. De evaluatie van dit type quasiquotes omvat twee stappen.

Een eerste stap is het evalueren van ieder element in de lijst. Analoog aan de redenering die in voorgaande sectie gevolgd werd, kan aangetoond worden dat het evalueren van ieder element in de lijst het juiste resultaat oplevert. Een tweede stap is dan het construeren van een lijst die de geëvalueerde expressies bevat.

Voor het aanmaken van een Scheme-lijst dient in SCALA-AM de store gebruikt te worden. Dit is nodig omdat er adressen in het heapgeheugen gealloceerd moeten worden. Vooraleer een waarde opgeslagen kan worden, moet er eerst een adres waarop de waarde opgeslagen kan worden aangemaakt worden. Dit adres kan dan gebruikt worden ter constructie van een pair en dus ter opbouw van een lijst. Dit is analoog aan manueel geheugenbeheer zoals bijvoorbeeld in `C`. Het alloceren van geheugen voor een waarde met bijvoorbeeld `malloc` komt overeen met het opslaan van een waarde in de store. Het resultaat hiervan is dan een pointer die naar de opgeslagen waarde wijst. Deze twee pointers worden dan tezamen in een pair opgeslagen.

3.3 Nesting en splicing

De huidige implementatie van quasiquoting ondersteunt voorlopig enkel niet-geneste vormen van quasiquoting en unquoting. Splicing wordt momenteel nog in geen enkele vorm ondersteund. De reden hiervoor is dat het implementeren van deze eigenschappen verregaande wijzigingen aan SCALA-AM zou vergen; dit aangezien de huidige implementatie van SCALA-AM voor enkele moeilijkheden zorgt. Het maken van de nodige wijzigingen hiervoor valt dan ook buiten de scope van deze thesis. In deze sectie worden de verwachte moeilijkheden toegelicht. Ook wordt beschreven hoe deze features mogelijk geïmplementeerd zouden kunnen worden. Aangezien deze moeilijkheden vrij complex zijn, wordt getracht enkel een hoog-niveau beeld van de meest significante problemen te geven. Ook de mogelijke oplossingen hiervoor zijn complex en zullen op eenzelfde manier besproken worden.

3.3.1 Geneste vormen van quasiquoting en unquoting

De R5RS Scheme-standaard beschrijft het gedrag van geneste quasiquotes als volgt [1]:

> *Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost backquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.*

Om te weten of een expressie al dan niet geëvalueerd mag worden, dient het nestingsniveau van de expressie binnen een quasiquote gekend te zijn, aangezien enkel componenten die zich op hetzelfde nestingsniveau als de buitenste quasiquote bevinden geëvalueerd mogen worden. Volgend voorbeeld verduidelijkt dit:

```
(cons 0 `(1 2 3 4 `(5 (+ 3 3) 7 ,@(list (+ 4 4) ,(+ 4 5)) 10 11) (+ 6 6) ,(+ 6 7)))
> (   0   1 2 3 4 `(5 (+ 3 3) 7 ,@(list (+ 4 4)     9   ) 10 11) (+ 6 6)     13   )
```

Iedere quasiquote verhoogt het nestingsniveau, iedere unquote verlaagt het nestingsniveau. De expressies die zich op hetzelfde nestingsniveau als de buitenste quasiquote bevinden zijn aangeduid in het groen en worden geëvalueerd. Alle andere expressies bevinden zich op een hoger nestingsniveau en worden dan ook niet geëvalueerd.

Om dit te kunnen implementeren werden twee mogelijke oplossingen uitgedacht. De opbouw van SCALA-AM blijkt er echter voor te zorgen dat deze oplossingsmethoden niet toepasbaar zijn. Het probleem dat zich hier stelt blijkt uit twee verwante deelproblemen te bestaan. Enerzijds blijkt de beperkte evaluatiecontext die SCALA-AM biedt problematisch. Anderzijds blijkt het feit dat gequote expressies niet naar Scheme-expressies gecompileerd worden een bijkomende moeilijkheid. Oplossingen voor het ene probleem blijken onmogelijk door het andere probleem en omgekeerd. In deze paragraaf proberen we een beeld van de problematiek te schetsen en ideeën voor oplossingen aan te reiken.

Evaluatie zonder context

Zoals besproken in secties 1.1.3 en 1.2.2, voert SCALA-AM de evaluatie van expressies uit met behulp van een stackmachine. De signatuur van de `stepEval`-methode uit de semantiek ziet er hierbij als volgt uit [13][16]:

```
/**
 * Defines what actions should be taken when an expression e needs to be
 * evaluated, in environment env with store store
 */
def stepEval(e: Exp, env: Environment[Addr], store: Store[Addr, Abs], t: Time):
Set[Action[Exp, Abs, Addr]]
```

De `stepEval`-methode beschikt enkel over de te evalueren expressie, de omgeving waarin deze expressie geevalueerd moet worden en de store. `StepEval` beschikt dus niet over het nestingsniveau van de expressie die geëvalueerd moet worden en kan dus onmogelijk bepalen of een expressie al dan niet geëvalueerd moet worden.

Om dit probleem te omzeilen zou er een extra parameter aan `stepEval` toegevoegd kunnen worden. Deze parameter houdt het nestingsniveau bij en wordt verhoogd bij het evalueren van een gequasiquote expressie en verlaagd bij het evalueren van een geünquote expressie. Expressies die zich op een positie bevinden waar het nestingsniveau groter dan nul is, dienen dan niet geëvalueerd te worden. Dit is echter te kort door de bocht aangezien er nagegaan moet worden of er binnen de expressies die niet geëvalueerd dienen te worden ergens een unquote staat. In dat geval kan het immers zijn dat er toch nog een deel van de expressie geëvalueerd dient te worden. Dit zorgt dan ook voor een eerste moeilijkheid die volgt uit het feit dat gequote expressies niet naar Scheme-expressies gecompileerd worden, hetgeen immers impliceert dat geneste quasiquotes en unquotes s-expressies, en dus geen Scheme-expressies, zijn. Dit is problematisch, aangezien de semantiek enkel Scheme-expressies kan evalueren. Dit probleem wordt uitgebreid in volgende paragraaf besproken.

Het aanpassen van de signatuur van de `stepEval`-methode zou bovendien een niet te verwaarlozen hoeveelheid werk met zich meebrengen, aangezien deze methode deel uitmaakt van de interface van een van de hoofdcomponenten van SCALA-AM, namelijk de semantiek.

Gequote expressies worden niet gecompileerd

In het bovenstaande voorbeeld wordt, zoals beschreven in sectie 3.2.2, het tweede argument van `cons`, namelijk de `quasiquote`-expressie `` `(1 2 3 4 `(5 (+ 3 3) 7 ,@(list (+ 4 4) ,(+ 4 5)) 10 11) (+ 6 6) ,(+ 6 7)) ``, geëvalueerd door ieder element uit de gequasiquote lijst te evalueren. Het vijfde element in de buitenste quasiquote, namelijk de expressie `` `(5 (+ 3 3) 7 ,@(list (+ 4 4) ,(+ 4 5)) 10 11) ``, is opnieuw een quasiquote. Aangezien dit element niet geünquoot is, wordt, zoals vermeld in sectie 3.1, deze expressie dan ook door de compiler gequoot. Dit zorgt er dan ook voor dat deze expressie een s-expressie blijft en niet naar een Scheme-expressie omgezet wordt. De evaluatie van de binnenste quasiquote loopt hier dus verkeerd, aangezien deze expressie als een gequote expressie aanzien wordt.

Een oplossing hiervoor kan zijn dat de gequote expressie, die dus een s-expressie is, gescand wordt. De scanmethode zou de s-expressie dan recursief doorlopen en houdt steeds het nestingsniveau bij. Wanneer dit niveau bijvoorbeeld nul wordt, kan de desbetreffende deelexpressie gecompileerd en geëvalueerd worden. Dit zorgt echter voor nieuwe moeilijkheden.

Een eerste probleem ontstaat uit het feit dat de evaluatie van een Scheme-expressie opnieuw een Scheme-expressie oplevert. Deze Scheme-expressie moet dan terug in de s-expressie gepast worden. Om dit te kunnen doen, moet er een methode voorzien worden die een Scheme-expressie naar een s-expressie omzet.

Een tweede probleem wordt veroorzaakt door het feit dat de evaluatie van een expressie via `stepEval` voltrokken wordt. Indien deze methode niet vanuit de scanmethode opgeroepen zou worden, is het moeilijk om te bepalen waar het resultaat van de evaluatie in de expressie ingevoegd moet worden. Een eerste mogelijke oplossing zou dus zijn om deze methode recursief op te roepen. Dit is echter in tegenstrijd met het designprincipe van SCALA-AM, dat bepaalt dat de machine de evaluatie stuurt. Bovendien is het resultaat van `stepEval` een verzameling van acties, en dus niet noodzakelijk een eindresultaat. Om van acties tot het eindresultaat te komen, zou een deel van de machinefunctionaliteit in de scanmethode geïntegreerd moeten worden. Dit zou echter de modulariteit van het framework ondergraven.

De scanmethode die beschreven wordt dient na te kunnen gaan op welk nestingsniveau een expressie zich bevindt. Hiervoor dient geweten te zijn op welk niveau deze methode opgeroepen wordt, wat een derde probleem voor de implementatie oplevert. Een gequote expressie die zich niet in een quasiquote bevindt, bevindt zich op een ander nestingsniveau dan een quote die zich wel in een quasiquote bevindt. De scanmethode zal echter nooit opgeroepen moeten worden op een nestingsniveau dieper dan één, aangezien een quasiquote die zich in een andere quasiquote bevindt ofwel geqoot ofwel geünquoot is. In het eerste geval dient de scanmethode opgeroepen te worden op nestingsniveau één, in het andere geval dient de scanmethode niet opgeroepen te worden aangezien de quasiquote niet door de compiler gequoot wordt. Desalniettemin kan de evaluator niet bepalen of de gequote expressie zich op nestingsniveau nul of één bevindt. Dit probleem wordt dus veroorzaakt door een gebrek aan evaluatiecontext, zoals besproken in voorgaande paragraaf.

Conclusie

In deze paragraaf werden twee problemen die de implementatie van geneste quasiquotes en unquotes bemoeilijken besproken. Deze problemen blijken met elkaar verweven te zijn. Door het combineren van de oplossingen voor beide deelproblemen, kan er een totaaloplossing bekomen worden. Door namelijk de evaluatiecontext uit te breiden en deze informatie te gebruiken om te bepalen of de scanmethode aangeroepen moet worden, kan geneste quasiquoting en unquoting geïmplementeerd worden.

Deze oplossing vereist echter een significante aanpassing van het framework. Niet alleen moet de interface van de semantiek aangepast worden, ook moet er een nieuwe compiler geschreven worden die Scheme-expressies naar s-expressies omzet, wat zeker niet triviaal is. Bovendien moet de `stepEval`-methode van de semantiek recursief aangeroepen worden, hetgeen extra moeilijkheden met zich mee zou brengen en het design van het

framework zou verslechteren. Het is dan ook de vraag of deze oplossingsstrategie uitvoerbaar en wenselijk is. Het is daarom best om, rekening houdende met de bemerkingen die in deze paragraaf aangehaald werden, verder naar andere, minder invasieve oplossingsstrategieën te zoeken. Het is hierbij aan te raden om na te gaan of een andere voorstellingswijze voor gequote expressies de besproken problemen kan verhelpen.

3.3.2 Splicing

De R5RS Scheme-standaard beschrijft het gedrag van `unquote-splicing` als volgt [1]:

> *If a comma appears followed immediately by an at-sign (@), then the following expression must evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the list are inserted in place of the comma at-sign expression sequence. A comma at-sign should only appear within a list or vector ‹qq template›.*

Het argument van `unquote-splicing` is dus een expressie die naar een lijst evalueert. Deze lijst wordt dan *opengebroken* en de elementen van de lijst worden rechtstreeks in de bovenliggende expressie geplaatst. Volgend voorbeeld geeft dit concreet weer:

```
(define (create-list from to)
  (if (> from to)
      '()
      (cons from
            (create-list (+ from 1)
                         to))))
`(0 1 2 ,@(create-list 3 9) 10)
> (0 1 2 3 4 5 6 7 8 9 10)
```

In dit voorbeeld is het resultaat van `(create-list 3 9)` de lijst `(3 4 5 6 7 8 9)`. Door de unquote-splicing worden de elementen van de lijst in de bovenliggende lijst ingevoegd. Wanneer we deze functionaliteit willen implementeren, houdt dit dan ook in dat er op een bepaald moment in de evaluatie meerdere resultaatwaarden ontstaan uit de evaluatie van één expressie. De evaluator is hier echter niet voor uitgerust. Zoals beschreven in sectie 1.1.2, wordt na het bereiken van een waarde de `stepKont`-functie van de semantiek opgeroepen. De signatuur van deze functie ziet er als volgt uit [13][16]:

```
/**
 * Defines what actions should be taken when a value v has been reached, and
 * the topmost frame is frame
 */
def stepKont(v: Abs, frame: Frame, store: Store[Addr, Abs], t: Time):
Set[Action[Exp, Abs, Addr]]
```

Deze functie krijgt de geëvalueerde waarde en het bovenste stackframe als argumenten mee en bepaalt hoe de evaluatie verdergezet moet worden. Hiervoor gebruikt het de geëvalueerde waarde; het frame geeft informatie over de context – de expressie – waarin de waarde gebruikt moet worden.

Van één naar meerdere waarden

Wanneer een unquote-splicing-expressie geëvalueerd wordt, is het argument van deze expressie reeds naar een lijst geëvalueerd. Deze lijst is het eerste argument van `stepKont`. Nu moet deze lijst nog gesplicet worden, wat impliceert dat er van één waarde naar meerdere waarden overgegaan moet worden.

Het probleem dat zich hierbij stelt is analoog aan het probleem dat in voorgaande sectie vastgesteld werd: er ontbreekt een evaluatiecontext. `stepKont` heeft namelijk geen enkele indicatie dat de waarde die het als argument meekrijgt gesplicet moet worden. Daarom zal `stepKont` deze waarde onveranderd laten alvorens verder te gaan met de evaluatie.

Het is aannemelijk om te denken dat het creëren van een speciaal stackframe voor unquote-splicing dit probleem zal verhelpen. Dit stackframe zou `stepKont` immers toelaten om te weten dat het argument resulteert uit de

evaluatie van een unquote-splicing en dus een lijst, die gesplicet moet worden, is. Frames op de stack bevatten echter enkel informatie over de expressie die ze vertegenwoordigen. Aangezien `unquote-splicing` slechts één argument heeft, zal een dergelijke frame geen nuttige informatie bevatten. Immers, de enige informatie die het frame kan bevatten is het argument dat geëvalueerd moet worden, maar de evaluator heeft ook informatie nodig over de bovenliggende lijst waarin het argument ingevoegd moet worden. Het toevoegen van informatie aan de stackframes, zoals deze bovenliggende lijst, om meer evaluatiecontext te bewaren is moeilijk. Stackframes worden aangemaakt vooraleer een deelexpressie geëvalueerd wordt, om de evaluatie verder te kunnen zetten na de evaluatie van de deelexpressie. Aangezien het resultaat van deze deelevaluatie onbekend is, kan nog niet geanticipeerd worden op hetgeen wat met het resultaat moet gebeuren, zoals bijvoorbeeld het uitvoeren van een splicing.

Aangezien de splicing niet uitgevoerd kan worden, kan `stepKont` in principe niets anders doen dan de geëvalueerde expressie terug te geven. In de volgende iteratie zal de machinecomponent van het framework `stepKont` opnieuw aanroepen, maar met het volgende stackframe. Dit stackframe bevat dan de context die in voorgaande stap ontbrak, maar op dit moment heeft de procedure geen enkele indicatie meer dat de waarde gesplicet moet worden. We moeten dus concluderen dat het toevoegen van het een speciaal stackframe voor unquote-splicing het probleem in dit geval niet verhelpt, maar enkel verschuift naar een verdere stap in de evaluatie. Bovendien kunnen we concluderen dat het gebruik van `stepKont` splicing, omwille van bovenstaande argumenten, onmogelijk maakt.

stepSplice

In voorgaande paragraaf werd verklaard waarom het gebruik van een extra stackframe in combinatie met `stepKont` het niet mogelijk maakt om splicing te implementeren. Dit impliceert dat de semantiek een nieuwe functionaliteit moet aanbieden die in staat is om de splicing uit te voeren. In het vervolg van dit verslag zullen we de naam `stepSplice` voor deze functionaliteit gebruiken.

De signatuur van `stepSplice` kan dezelfde zijn dan deze van `stepKont`:

```
def stepSplice(list: Abs, frame: Frame, store: Store[Addr, Abs], t: Time):
Set[Action[Exp, Abs, Addr]]
```

`stepSplice` zal dan de splicing uitvoeren gegeven de te splicen lijst en een frame dat de nodige context voor de splicing bevat. Voor ieder frametype bepaalt `stepSplice` dan hoe de splicing uitgevoerd moet worden. Frames bevatten immers informatie over de huidige staat van de evaluatie. Hieruit kan dan ook afgeleid worden waar de waarden die uit de splicing resulteren ingevoegd moeten worden. Als resultaat moet `stepSplice` dan, net zoals `stepEval` en `stepKont`, acties aan de machine teruggeven die aangeven hoe de evaluatie verdergezet moet worden.

Voor deze strategie is echter wél een speciaal stackframe voor unquote-splicing nodig. Na het evalueren van het argument van de unquote-splicing, moet `stepKont` er immers voor zorgen dat `stepSplice` aangeroepen wordt. Dit kan enkel als `stepKont` weet dat de geëvalueerde expressie gesplicet moet worden. Door middel van een nieuwe actie die dan gedefinieerd wordt, kan de machine dan geïnstrueerd worden op de `stepSplice`-methode op te roepen.

Bovenstaande redenering zou een mogelijke implementatiestrategie kunnen zijn. Er duiken echter enkele problemen op. Eerst en vooral is het niet meteen duidelijk hoe er voor sommige soorten stackframes een splicing kan gedefinieerd worden. Dit hangt samen met een tweede probleem, dat veroorzaakt wordt door het nesten van expressies: niet alle expressies in SCALA-AM zijn Scheme-expressies. Indien splicing in een s-expressie voorkomt, moet ook dit correct geëvalueerd kunnen worden. Deze problematiek werd reeds uitvoerig in sectie 3.3.1 besproken en wordt hier daarom niet herhaald. Het toevoegen van splicing zou de daar geadviseerde oplossing bovendien nog kunnen compliceren.

3.4 Conclusie

In deze paragraaf werd besproken hoe een eenvoudige vorm van quasiquoting geïmplementeerd werd. Tevens werd er verklaard waarom geneste quasiquotes en unquotes alsook splicing nog niet geïmplementeerd zijn. Hiervoor werden de meest prangende problemen toegelicht. Er werden ook enkele mogelijke oplossingen geschetst. Het is echter te voorzien dat er ook nog andere problemen, die niet in deze paragraaf besproken werden, bestaan.

De meeste problemen vinden hun oorzaak in de voorstelling van gequote waarden in SCALA-AM. Aangezien deze niet naar een Scheme-expressie gecompileerd worden, maar s-expressies blijven, moet er bij de evaluatie van geneste quasiquotes, unquotes, en unquote-splicings met deze twee representaties rekening gehouden worden. Ook de beperkte evaluatiecontext als gevolg van het gebruik van een stackmachine waarvan de frames slechts beperkte informatie bevatten zorgt voor een bijkomende moeilijkheid. Het toevoegen van extra informatie aan deze frames blijkt eveneens geen mogelijke optie.

Voor ieder probleem werd er getracht een mogelijke oplossing te formuleren. Voor geneste quasiquotes en unquotes is het onvermijdelijk om de evaluatiecontext die de interpreter gebruikt aan te passen. Voor splicing stellen we voor om een nieuwe methode, `stepSplice`, aan de semantiek toe te voegen. De oplossingsmethoden die in deze paragraaf besproken werden zijn natuurlijk niet uniek; voor ieder probleem bestaan er zoals steeds meerdere mogelijke oplossingen. Elk van deze oplossingen zal echter wel steeds een antwoord op de beschreven problemen moeten bieden.

4 Benchmarking van SCALA-AM

Een derde deel van de bachelorproef houdt het benchmarken van de Scheme-implementatie van SCALA-AM in. Het doel is hierbij om op een kwantitatieve manier na te gaan in hoeverre met behulp van de huidige implementatie realistische R5RS-programma's geëvalueerd en geanalyseerd kunnen worden en of er eventueel bugs in de implementatie aanwezig zijn. Met de huidige implementatie wordt de versie na de uitvoering van deze bachelorproef bedoeld. Bovendien willen we nagaan welke uitbreidingen van Scheme binnen SCALA-AM het meest nuttig zijn door te bepalen welke features en primitieven van Scheme het meest gebruikt worden in realistische programma's.

Om te kunnen benchmarken is er nood aan referenties die voor het beoordelen van de implementatie gebruikt kunnen worden. Hiervoor werden Scheme-programma's en codefragmenten verzameld. Om een zo representatief mogelijk beeld van het gebruik van R5RS-features te bekomen, werden er hiervoor verschillende bronnen gebruikt:

- Codebestanden van VUB-cursussen
    - *Algoritmen & Datastructuren 1* [2]
    - *Structuur van Computerprogramma's 1* [9]
    - *Principles of Object-Oriented programming Languages* [10]
- Benchmarksuites en documentatie van reeds bestaande Scheme-implementaties
    - *Chez Scheme* [5]
    - *Gambit* [6]
    - *MIT/GNU Scheme* [8]
    - *Sigscheme* [12]
- Scheme-documentatie [1][3]
- Online codebases [4][7][11].

Voor deze bachelorproef werden 125 benchmarks aan het framework toegevoegd, om zo een totaal van 168 benchmarks te bekomen. Voor iedere benchmark werd nagegaan of deze door SCALA-AMs concrete interpreter correct geëvalueerd wordt. Het gebruik van de concrete interpreter is eenvoudiger dan het gebruik van de abstracte interpreter en heeft geen invloed op de resultaten. Indien een benchmark niet correct geëvalueerd wordt, wordt nagegaan wat de oorzaak hiervan is. In dat geval zeggen we dat de benchmark faalt; van een benchmark die wel correct geëvalueerd wordt, zeggen we dat hij slaagt. Verderop in dit verslag wordt besproken hoe deze benchmarks verzameld werden.

De meest voorkomende oorzaak van een benchmark die niet correct geëvalueerd wordt, is het ontbreken van een Scheme-feature in de semantiek van SCALA-AM. Scheme's features kunnen in twee categorieën onderverdeeld worden. Een eerste categorie zijn de reguliere primitieven. Deze primitieven stellen gewone procedures voor, zoals `+` en `char?`. De applicatie van een dergelijke primitieven verloopt steeds op dezelfde manier. Een tweede categorie bestaat uit primitieven die een bijzondere functie, zoals bijvoorbeeld Input/Output, hebben en de zogenaamde *special forms*. Special forms zijn procedures die een speciale evaluatiemethode vereisen en eveneens voor speciale Scheme-features, zoals controlestructuren, gebruikt worden [1]. Deze features zullen in het vervolg van dit verslag taalfeatures genoemd worden. Voorbeelden van dergelijke primitieven en special forms zijn `if`, `quasiquote`, `eval`, `apply` en `open-input-port`.

De volledige resultaten van het benchmarken zijn in bijlage A terug te vinden. In de volgende secties van dit verslag worden deze resultaten beknopt beschreven. Eerst wordt nagegaan in hoeverre de primitieven voor verschillende taalfeatures van Scheme ondersteund worden. Daarna wordt hetzelfde voor de reguliere primitieven van Scheme onderzocht. Hiervoor zullen we telkens nagaan hoeveel benchmarks er door het ontbreken van een bepaalde feature falen. Aangezien de implementatie van Scheme in het framework nog vrij onvolledig is, is het mogelijk dat een benchmark omwille van meerdere redenen faalt. Er kunnen dan bijvoorbeeld meerdere primitieven ontbreken. In dat geval wordt de benchmark bij alle oorzaken meegerekend.

Om dit te vereenvoudigen, zullen de primitieven en special forms telkens in verschillende categorieën onderverdeeld worden. Op deze manier kan er per categorie nagegaan worden hoeveel benchmarks er falen door het

ontbreken van primitieven en special forms uit de desbetreffende categorie. Als er een aantal benchmarks omwille van het ontbreken van primitieven en special forms uit een bepaalde categorie falen, zeggen we dat er in die categorie benchmarks falen of dat een categorie aan de oorzaak de falende benchmarks ligt.

Nadien wordt nagegaan welke andere oorzaken ervoor zorgen dat sommige benchmarks niet correct geëvalueerd en geanalyseerd kunnen worden. Tot slot worden de verkregen resultaten gebruikt om te bepalen welke uitbreidingen aan het framework het best eerst uitgevoerd kunnen worden om zo snel mogelijk een zo groot mogelijk aantal benchmarks correct te kunnen evalueren.

Verzamelen van benchmarks

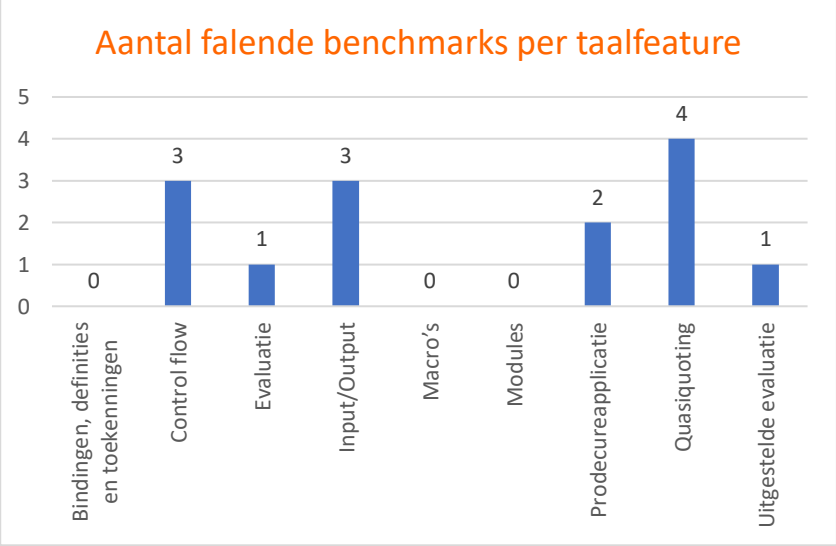
Zoals reeds vermeld werden benchmarks uit verschillende bronnen verzameld. Verschillende cursussen die aan de Vrije Universiteit Brussel gedoceerd worden gebruiken de taal Scheme. De bronbestanden van deze cursussen kunnen dan ook gebruikt worden voor zover deze de R5RS-standaard volgen en er geen definities uit andere bestanden geïmporteerd moeten worden. Dit laatste is immers niet mogelijk omdat SCALA-AM geen modules ondersteunt. Ook reeds bestaande Scheme-implementaties bevatten vaak al benchmarks die ook in SCALA-AM gebruikt kunnen worden. Tot slot kunnen ook andere bronbestanden uit bijvoorbeeld online codebases gebruikt worden.

De uitvoering van iedere benchmark moet een resultaat opleveren dat gebruikt kan worden om na te gaan of de benchmark correct werkt. Aan bronbestanden die geen resultaat opleveren na evaluatie werden extra statements toegevoegd zodat er toch een resultaat verkregen wordt.

4.1 Primitieven en special forms voor taalfeatures

Om eenvoudig na te kunnen gaan welke taalfeatures er het beste aan het framework toegevoegd worden, zullen we de primitieven en special forms die bij dergelijke features horen eerst classificeren. Deze classificatie kan opgesteld worden volgens de taalfeature waarvoor de desbetreffende primitieve of special form gebruikt wordt. Voor deze bachelorproef zullen we negen verschillende taalfeatures onderscheiden [17].

Het ontbreken van bepaalde primitieven en special forms zal ervoor zorgen dat een benchmark faalt. In Figuur 4 wordt voor iedere taalfeature weergegeven hoeveel benchmarks falen omwille van problemen met deze feature. Hierbij vallen verschillende dingen op.



Figuur 4 *Aantal falende benchmarks per taalfeature*

Er zijn drie taalfeatures die niet aan de oorzaak van een niet-werkende benchmark liggen. Dit kan betekenen dat alle bijbehorende primitieven en special forms geïmplementeerd zijn, dat de primitieven en special forms die tot de taalfeature behoren niet door de benchmarks gebruikt worden of dat slechts een enkele van de bij de taalfeature horende primitieven en special forms werken, maar dat dit de enige zijn die door de benchmarks gebruikt worden.

De eerste categorie, *Bindingen, definities en toekenningen*, omvat special forms zoals `define`, `let` en `set!`. Aangezien alle special forms in deze categorie geïmplementeerd zijn, is dit resultaat logisch. De andere twee features waarbij geen enkele benchmark faalt, namelijk macro's en modules, zijn nog niet geïmplementeerd. Aangezien er geen enkele benchmark is die deze features lijkt te gebruiken, kunnen we besluiten dat deze taalfeatures niet dringend geïmplementeerd moeten worden. Bovendien is het niet eenvoudig om de categorie *Modules* te implementeren, aangezien deze niet door de R5RS-standaard vastgelegd wordt [1].

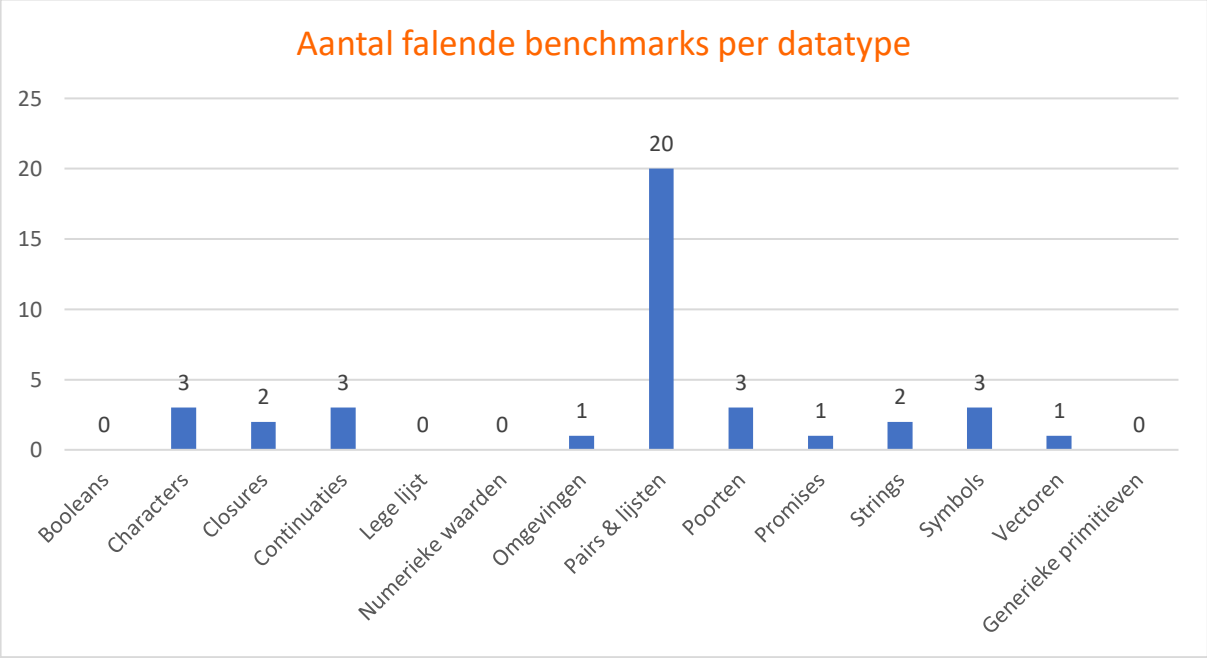
De andere taalfeatures zijn nog niet volledig geïmplementeerd. Dit zorgt er voor dat er verschillende benchmarks niet correct geëvalueerd kunnen worden. Een krachtige control-flowprimitieve uit Scheme die ontbreekt is bijvoorbeeld `call-with-current-continuation`. Ook de `eval`-primitieve en de `apply`-primitieve, die respectievelijk tot de categorieën *Evaluatie* en *Procedureapplicatie*, behoren ontbreken. De Scheme-implementatie van SCALA-AM heeft bovendien nog geen primitieven voor *Input/Output*, waardoor alle benchmarks die hiervan gebruikmaken falen. Ook uitgestelde evaluatie, waarvoor de `delay` special form en de `force`-primitieve gebruikt worden, wordt nog niet ondersteund.

Het grootste aantal falende benchmarks wordt bij de categorie *Quasiquoting* genoteerd. Nochtans werd er in sectie 3 van dit rapport uitgebreid besproken hoe een basisversie van quasiquoting in SCALA-AM geïmplementeerd werd. Aangezien quasiquoting niet volledig geïmplementeerd werd, is het dus logisch dat benchmarks die van de niet-geïmplementeerde onderdelen en primitieven gebruikmaken falen. Het hoge aantal is te verklaren doordat er voor de implementatie van quasiquoting extra benchmarks die hiervan gebruikmaken toegevoegd werden. Dit zorgt er dan ook voor dat de grafiek een vertekend beeld hieromtrent weergeeft.

4.2 Reguliere primitieven

Het overgrote deel van de primitieven in Scheme zijn reguliere primitieven. Om een beknopt overzicht te kunnen geven van de primitieven die in de Scheme-implementatie ontbreken, zullen deze primitieven eerst geclassificeerd worden naargelang het datatype waarop ze werkzaam zijn [17]. Voor ieder datatype kan er dan nagegaan worden hoeveel benchmarks er in de daarbij horende categorie falen. Op deze manier kan kwantitatief vastgesteld worden welke datatypes meer ondersteund moeten worden.

Figuur 5 geeft per datatype weer hoeveel benchmarks er falen omwille van ontbrekende primitieven die op dat datatype werkzaam zijn.



Figuur 5 *Aantal falende benchmarks per datatype*

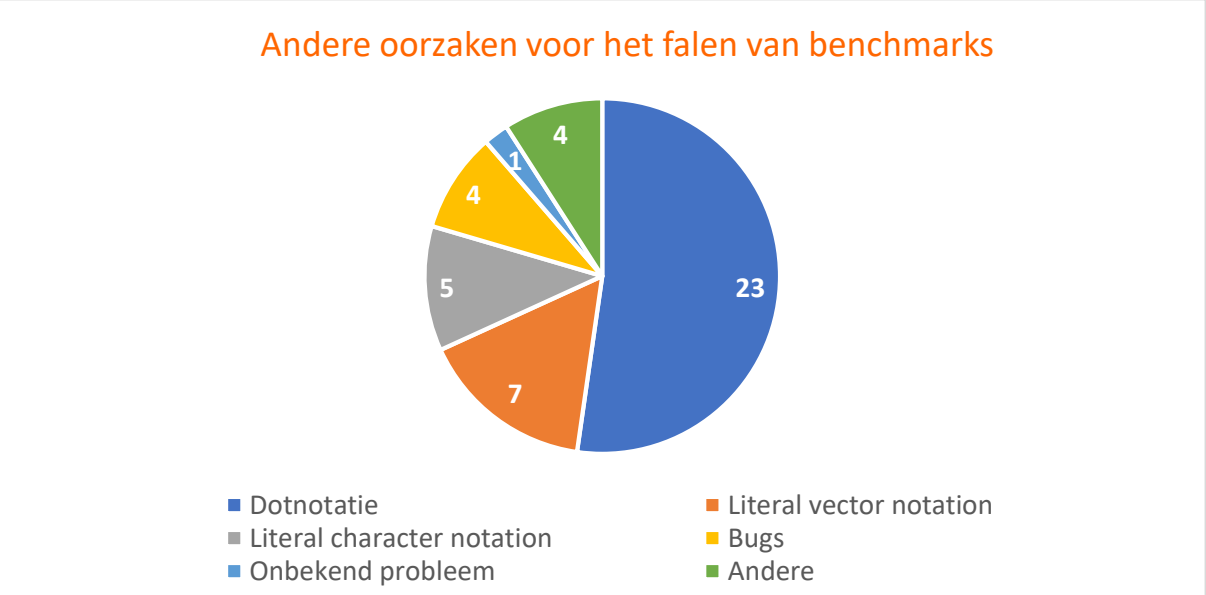
Voor verschillende datatypes ontbreken er geen primitieven die in de benchmarks gebruikt worden. Voor enkele eenvoudige datatypes, zoals booleans en de lege lijst, is dit logisch aangezien de R5RS-standaard weinig primitieven beschrijft die op deze datatypes werken. Voor andere datatypes is er een verband met de resultaten uit sectie 4.1, aangezien de primitieven van sommige taalfeatures op bepaalde datatypes werkzaam zijn. Zo worden promises voor uitgestelde evaluatie gebruikt. Het is dan ook logisch dat in Figuur 5 dezelfde waarde bij *promises* aangegeven staat dan in Figuur 4 bij *uitgestelde evaluatie*. Andere verbanden zijn bijvoorbeeld de `eval`-primitieve die op zogenaamde Scheme-omgevingen werkzaam is en de `call-with-current-continuation`-primitieve die een continuatie als argument neemt.

Naast de datatypes waarvoor er geen primitieven ontbreken, zijn er een groot aantal datatypes waarbij er een aantal benchmarks falen. Dit komt dan omdat er bepaalde primitieven die door de benchmarks gebruikt worden ontbreken. Het grootste aantal problemen doet zich in de categorie *Pairs & lijsten* voor, wat ervoor zorgt dat meer dan tien procent van de benchmarks niet werkt. Het is logisch dat deze categorie problematisch is. Enerzijds vindt Scheme zijn oorsprong in Lisp, waarin pairs, en dus lijsten, een belangrijke rol spelen. Ook in Scheme is dit datatype erg belangrijk. Een tweede oorzaak is dat Scheme verschillende krachtige en veelgebruikte hogere-ordefuncties die op lijsten werken, zoals `map` en `for-each`, heeft. In SCALA-AM is het echter enorm complex om dergelijke primitieven te implementeren, waardoor deze veelal nog niet geïmplementeerd zijn. Ook andere veelgebruikte functies die op lijsten werken, zoals `reverse`, zijn nog niet geïmplementeerd.

De `append`-primitieve, waarmee twee lijsten geconcateneerd kunnen worden, bleek ook een veelgebruikte primitieve. Daarom werd deze reeds tijdens deze bachelorproef toegevoegd. In sectie 2.3 van dit rapport wordt de implementatie van `append` besproken.

4.3 Andere oorzaken van falende benchmarks

Naast het ontbreken van reguliere primitieven, primitieven van taalfeatures en special forms kunnen ook andere mogelijke oorzaken aan de basis van falende benchmarks liggen. Figuur 6 geeft voor verschillende redenen weer hoeveel benchmarks ten gevolge van deze redenen falen. In deze paragraaf worden de belangrijkste problemen toegelicht.



Figuur 6 *Andere oorzaken voor het falen van benchmarks*

4.3.1 Functies met een variabel aantal argumenten

Een krachtige taalfeature van Scheme waar geen primitieven voor bestaan, zijn functies met een variabel aantal argumenten. Met deze taalfeature is het mogelijk om functies te definiëren waarvan het aantal argumenten op voorhand niet gekend is. In Scheme wordt hiervoor een speciale dotnotatie gebruikt [1]. We geven een voorbeeld:

```
; De procedure sum neemt minstens 2 argumenten a en b.
; Eventuele overige argumenten worden in een lijst args opgeslagen.
(define (sum a b . args)
  (+ a
     b
     (apply + args)))
(sum 1)
> ERROR - sum: arity mismatch
(sum 1 2)
> 3
```

```
(sum 1 2 3 4 5)
> 15

; Deze procedure som neemt een willekeurig aantal argumenten.
; Eventuele argumenten worden in een lijst args opgeslagen.
(define (som . args)
  (apply + args))
(som 1)
> 1
(som 1 2)
> 3
(som 1 2 3 4 5)
> 15
```

Meer dan 20 benchmarks maken van deze notatie gebruik, hoewel deze niet door het framework ondersteund wordt. Om deze feature te ondersteunen, moeten onder andere de parser en de semantiek van het framework aangepast worden.

4.3.2 Literal notations

Elementen van sommige datatypes kunnen op een verkorte manier in een programma neergeschreven worden. Zo wordt de notatie `#\` gebruikt om een character te noteren en wordt een gelijkaardige notatie voor vectoren gebruikt. Aangezien deze notaties nog niet door de parser ondersteund worden, zal iedere benchmark die hiervan gebruikmaakt falen. Net zoals voor functies met een variabel aantal argumenten, wordt ook in de literal notation voor pairs van een dot gebruikgemaakt.

4.3.3 Bugs

Tijdens het benchmarken werden er verschillende bugs in de implementatie ontdekt. Deze bugs hebben voornamelijk met de abstracte interpretatie van programma's te maken. Voor deze bachelorproef werden alle benchmarks op de concrete interpreter van SCALA-AM uitgevoerd. We verwachten dan ook telkens dat het resultaat van iedere benchmark één concrete waarde is. Dit blijkt niet altijd het geval, hetgeen op een bug duidt. Ook een foutieve uitkomst van een benchmark kan het gevolg van een bug zijn. Sommige bugs die tijdens het benchmarken gevonden werden, konden reeds door andere SCALA-AM-ontwikkelaars opgelost worden.

4.4 Toekomstige uitbreidingen

Vertrekkend van bovenstaande resultaten kunnen we nu bepalen welke uitbreidingen van Scheme in SCALA-AM het meest noodzakelijk zijn en toelaten om zo snel mogelijk een groot aantal programma's te evalueren en te analyseren. Aangezien het mogelijk is dat een benchmark faalt omwille van verschillende ontbrekende primitieven en functionaliteiten, zal de implementatie van één extra feature mogelijk niet veel extra benchmarks werkend maken. Echter, het aantal voorkomens van ieder probleem, zoals bijvoorbeeld een ontbrekende primitieve, geeft een goede indicatie van de belangrijkste niet-ondersteunde Scheme-features.

4.5 Conclusie

De meest voorkomende niet-ondersteunde Scheme-feature is de dotnotatie voor functies met een variabel aantal argumenten. We besluiten dan ook dat deze feature best zo snel mogelijk aan het framework toegevoegd dient te worden. Ook de (hogere-orde-)functies die op pairs en lijsten werken, zoals `map`, `for-each` en `reverse`, worden vaak in de benchmarks gebruikt. Het is dan ook aan te raden om ook deze primitieven eerst aan het framework toe te voegen alvorens andere uitbreidingen te maken.

5 Conclusie

In dit rapport werd de uitvoering van de bachelorproef besproken. Deze bachelorproef had als doel om SCALA-AM uit te breiden om zo meer Scheme-programma's die aan de R5RS-standaard voldoen te kunnen analyseren.

Eerst werd besproken hoe enkele wiskundige primitieven aan het framework toegevoegd werden, waarvoor zowel de lattices als de semantiek van het framework uitgebreid dienden te worden. Het toevoegen de `append`-primitieve was complexer aangezien deze het gebruik van de store vereist. Hierbij werd ook besproken hoe het design van het framework de implementatie bemoeilijkte.

Daarna werd besproken hoe de implementatie van quasiquoting gebeurde. Hierbij werden enkele problemen, zoals het gebrek aan evaluatiecontext en de voorstelling van gequote waarden, beschreven. Deze problemen zorgen ervoor dat het moeilijk is om quasiquoting volledig te ondersteunen. Oplossingen voor het ene probleem blijken door het andere probleem onmogelijk en vice versa.

Tot slot werden de resultaten van het benchmarken besproken, hetgeen toelaat om na te gaan welke features best eerstvolgend aan het framework toegevoegd kunnen worden. Uit de resultaten blijkt dat de ondersteuning van de dotnotatie voor functies met een variabel aantal argumenten en de implementatie van enkele primitieven die op lijsten werkzaam zijn best eerst toegevoegd kunnen worden.

In deze bachelorproef werd er dus voor gezorgd dat er meer Scheme-programma's door het framework geanalyseerd kunnen worden dan voorheen. De resultaten van het benchmarken laten andere SCALA-AM-ontwikkelaars toe om nu op een gerichte manier nieuwe features aan het framework toe te voegen. Bovendien werd een overzicht gegeven van de problemen die mogelijk te verwachten zijn bij de implementatie van sommige features en werd aangegeven hoe deze problemen mogelijk omzeild kunnen worden. Op deze manier draagt deze bachelorproef ook bij aan de toekomstige uitbreidingen van het framework, aangezien ze implementatiedoelstellingen en -problemen vastlegt.

A Overzicht van de benchmarkingsresultaten

Deze bijlage bevat een overzicht van de volledige benchmarkingsresultaten. Eerst worden de resultaten per benchmark beschreven. Daarna wordt dit voor iedere ontbrekende primitieve, special form en language feature gedaan, waarna de ook de onopgeloste bugs besproken worden. Daarna wordt kort een algemeen overzicht van de resultaten gegeven. Tot slot worden de reeds verholpen problemen aangeduid.

a Resultaten per benchmark

In deze sectie wordt voor iedere benchmark aangegeven of deze slaagt of faalt wanneer uitgevoerd met de concrete interpreter van SCALA-AM. Indien een benchmark faalt, kan afgelezen worden wat de oorzaak hiervan is. Sommige benchmarks werden licht aangepast om ze toch met de huidige implementatie te laten werken. Deze benchmarks worden in de tabellen als werkend aangeduid, maar er wordt wel bij vermeld welke functionaliteit ontbreekt om de oorspronkelijke versie te laten werken.

De resultaten in deze tabellen laten zien dat sommige problemen vooral bij benchmarks uit eenzelfde bron voorkomen. Zo worden de dotnotatie en de literal vector notation voornamelijk in benchmarks van *Algoritmen & Datastructuren 1* gebruikt.

De benchmarks die omwille van de uitbreidingen in deze bachelorproef werkend zijn gemaakt worden in het groen aangeduid. Dit aantal is zeer gering, omwille van het grote aantal ontbrekende Scheme-features in SCALA-AM. Het implementeren van enkele ontbrekende primitieven, zoals in deze bachelorproef gedaan werd, zal dan ook weinig benchmarks werkende maken. Vaak zijn er immers meerdere problemen bij één bepaalde benchmark, zodat deze allemaal opgelost zouden moeten kunnen worden vooraleer de benchmark werkt.

i Benchmarks van Algoritmen & Datastructuren 1

Naam benchmark	OK?	Ontbrekende primitieven	Ontbrekende language features
abstrct.scm	✓		
bfirst.scm			Dotnotatie
bst.scm			Dotnotatie
bubsort.scm	✓		Literal vector notation
dict.scm			Dotnotatie
graf.scm			Dotnotatie
heap.scm	✓		Literal vector notation
inssort.scm	✓		Literal vector notation
linear.scm			Dotnotatie
list.scm			Dotnotatie
mesort.scm			Literal vector notation, Bug #1
prioq.scm	✓		Literal vector notation
qsort.scm	✓		Literal vector notation
qstand.scm	✓		
queue.scm			Dotnotatie
quick.scm	✓		
RBtreeADT.scm			Dotnotatie
selsort.scm			Literal vector notation
stack.scm	✓		
stspaceCODE.scm		`reverse`	Dotnotatie

ii Benchmarks van Gambit

Naam benchmark	OK?	Ontbrekende primitieven	Ontbrekende language features
array1.scm	✓		
browse.scm		`string-ref`	Literal character notation
cat.scm		`write-char`, `read-char`, `eof-object?`,	Input/Output

Naam benchmark	OK?	Ontbrekende primitieven	Ontbrekende language features
		`close-output-port`, `close-input-port`, `open-output-file`, `open-input-file`	
compiler.scm		Veel niet-geïmplementeerde primitieven	Dotnotatie, Literal character notation
ctak.scm		`call-with-current-continuation`	Continuaties
deriv.scm		`map`	
destruc.scm	✓		
diviter.scm	✓		
earley.scm		`map`	
fibc.scm		`call-with-current-continuation`	Continuaties
graphs.scm			
lattice.scm		`map`	
matrix.scm	✓		
mazefun.scm	✓		
nboyer.scm		Veel niet-geïmplementeerde primitieven	Dotnotatie
paraffins.scm	✓		
perm9.scm	✓		
peval.scm		`eval`, `apply`	Evaluatie, functieapplicatie
primes.scm	✓		
puzzle.scm		`for-each`	Continuaties
sboyer.scm			Dotnotatie
scheme.scm		Veel niet-geïmplementeerde primitieven	Dotnotatie
slatex.scm		`map`	Literal character notation
string.scm		`substring`	
sum.scm	✓		
sumloop.scm	✓		
tail.scm		`reverse`, `list->string`, `char=?`, `read-char`, `eof-object?`, `close-output-port`, `close-input-port`, `open-output-file`, `open-input-file`	Input/Output, literal character notation
tak.scm	✓		
trav1.scm			Unquote-splicing
triangl.scm		`list->vector`, `vector->list`	
wc.scm		`char=?`, `read-char`, `eof-object?`, `close-input-port`, `open-output-file`	Input/Output, literal character notation

iii Benchmarks van Sigscheme

Naam benchmark	OK?	Ontbrekende primitieven	Ontbrekende language features
takr.scm	✓		
rec.scm	✓		
mem.scm	✓		
loop.scm	✓		
let-loop.scm	✓		
case.scm			Bug #1
arithint.scm	✓		

iv Benchmarks van Structuur van Computerprogramma's 1

Naam benchmark	OK?	Ontbrekende primitieven	Ontbrekende language features
2.1.scm	✓		
2.4.scm	✓		
3.1.scm	✓		
3.2.1.scm	✓		
3.2.scm	✓		
3.3.scm			
3.4.scm	✓		
3.6.scm		`for-each`	
3.8.scm	✓		
3.9.scm			Stackoverflow tijdens uitvoering
4.1.scm	✓		
4.8.scm			Bug #2
5.14.3.scm	✓		
5.19.scm	✓		
5.20.4.scm	✓		
5.21.scm		`reverse`	
5.22.scm		`apply`	Prodecureapplicatie
5.6.scm	✓		
5.7.scm		`reverse`	
7.11.scm	✓		
7.12.scm	✓		
7.13.scm		`for-each`	
7.14.scm	✓		
7.15.scm		`map`	
7.16.scm	✓		
7.17.scm	✓		
7.18.scm		`reverse`	
7.2.scm	✓		
7.3.scm	✓		
7.4.scm	✓		
7.5.scm			Dotnotatie
7.6.scm			Dotnotatie
7.9.scm			Dotnotatie
8.1.1.scm	✓		
8.1.3.scm	✓		
8.10.scm	✓		
8.11.scm		`map`, `reverse`	Dotnotatie
8.12.scm	✓		
8.13.scm	✓		
8.15.scm	✓		
8.16.scm		`for-each`	Dotnotatie
8.5.scm			Dotnotatie
8.6.scm	✓		
9.12.scm	✓		
9.13.scm			Bug #1
9.14.scm	✓		
9.15.scm	✓		
9.16.scm	✓		
9.17.scm	✓		
9.18.scm	✓		
9.2.scm	✓		
9.3.scm	✓		

Naam benchmark	OK?	Ontbrekende primitieven	Ontbrekende language features
9.5.scm	✓		
9.6.scm	✓		
9.7.scm	✓		
9.8.scm	✓		
9.9.scm	✓		

v Benchmarks uit andere bronnen

Naam benchmark	OK?	Ontbrekende primitieven	Ontbrekende language features
ack.scm	✓		
append.scm	✓		
blur.scm	✓		
bound-precision.scm	✓		
boyer.scm			Uitvoering stopt niet
church.scm	✓		
church-0.scm	✓		
church-1.scm	✓		
church-2.scm	✓		
church-2-num.scm	✓		
church-6.scm	✓		
collatz.scm	✓		
conform.scm		`map`, `symbol->string`	
conform2.scm		`map`, `symbol->string`	Dotnotatie
count.scm	✓		
cpstak.scm	✓		
dderiv.scm	✓		
divrec.scm	✓		
easter.scm	✓		
eta.scm	✓		
fact.scm	✓		
fib.scm	✓		
gcipt.scm	✓		
grid.scm			Bug #1
inc.scm	✓		
infinite-1.scm	✓		
infinite-2.scm	✓		
infinite-3.scm	✓		
kcfa2.scm	✓		
kcfa3.scm	✓		
letrec-begin.scm	✓		
loop2.scm	✓		
looping.scm			
mceval.scm	✓		
mj09.scm	✓		
mut-rec.scm	✓		
nqueens.scm	✓		
primtest.scm	✓		
quadractic.scm	✓		
quasiquoting.scm			Geneste unquoting, dotnotatie, unquote-splicing
quasiquoting-simple.scm	✓	`reverse`	
regex.scm	✓		
rsa.scm	✓		

Naam benchmark	OK?	Ontbrekende primitieven	Ontbrekende language features
sat.scm	✓		
scm2c.scm		`for-each`	Geneste unquoting, unquote-splicing
scm2java.scm	✓		
SICP-compiler.scm			Geneste unquoting, dotnotatie
sq.scm	✓		
Streams.scm		`delay`, `force`	Dotnotatie, uitgestelde evaluatie
sym.scm	✓		
takl.scm	✓		
widen.scm	✓		
work.scm	✓		

b Voorkomens per probleem

In deze sectie wordt aangeduid hoeveel keer iedere ontbrekende feature of primitieve voorkomt. Bovendien worden ook de aanwezige bugs besproken.

i Primitieven en taalfeatures

Ontbrekende primitieve	Aantal voorkomens	Ontbrekende taalfeature	Aantal voorkomens
`map`	8	**Geneste quasiquoting**	0
`for-each`	5	**Geneste unquoting**	3
`reverse`	7	**Unquote-splicing**	3
`list->string`	1	**Dotnotatie**	23
`list->vector`	1	**Uitgestelde evaluatie**	1
`vector->list`	1	**Continuaties**	3
`char=?`	3	**Input/Output**	3
`substring`	1	**Evaluatie**	1
`string-ref`	1	**Prodecureapplicatie**	2
`write-char`	1	**Literal character notation**	5
`read-char`	3	**Literal vector notation**	7
`close-output-port`	2		
`eof-object?`	3		
`close-input-port`	3		
`open-output-file`	3		
`open-input-file`	2		

ii Bugs

In de resultaten in sectie a werden twee bugs vermeld, die respectievelijk de nummers 1 en 2 kregen.

Bug #1

Deze bug heeft betrekking op een probleem met de concrete evaluatie. Tijdens een dergelijke evaluatie wordt er met concrete waarden gewerkt, waardoor de precisie maximaal is. SCALA-AM gedraagt zich dan ook als een concrete interpreter. Echter, bij de benchmarks waar dit probleem zich voordoet geeft de machine een error. Deze kreeg namelijk van de semantiek meerdere acties terug, wat niet mag tijdens een concrete evaluatie. Immers, er kan geen ambiguïteit zijn aangezien alle waarden exact gekend zijn. Deze bug wijst er dan ook op dat er zich binnen de semantiek ergens precisieverlies voordoet.

Bug #2

Deze bug doet zich voor bij benchmark `4.8.scm`. De evaluatie van de benchmark resulteert in een verkeerd resultaat. Dit is mogelijk het gevolg van precisieverlies bij het rekenen.

c Algemeen overzicht van de benchmarkingsresultaten

Totaal aantal benchmarks: 168
Aantal werkende benchmarks: 106 (≈ 63%)
Aantal niet-werkende benchmarks: 62
Aantal voorkomens van bugs: 4
Totaal aantal vastgestelde problemen: 110

d Overzicht van de reeds verholpen problemen

De tijdens het benchmarken verzamelde resultaten worden reeds door de SCALA-AM-ontwikkelaars gebruikt als leidraad voor het verder uitbreiden van Scheme in SCALA-AM. Hierdoor konden reeds enkele belangrijke problemen verholpen werden. In deze paragraaf worden de reeds verholpen problemen aangeduid.

i Reeds geïmplementeerde primitieven, special forms en taalfeatures

Onderstaande tabel geeft aan welke primitieven, special forms en taalfeatures reeds geïmplementeerd werden en hoe vaak deze in de benchmarkingsresultaten als ontbrekend aangeduid werden.

Geïmplementeerde primitieve/special form	**Aantal voorkomens**	**Geïmplementeerde taalfeature**	**Aantal voorkomens**
`do`	13	**Named let**	13
`member`	3		
`memq`	3		
`assq`	1		
`append` (zie sectie 2.3)	14		
`list-ref`	1		
`string->symbol`	1		
`string<?`	1		
`exact->inexact`	2		
`remainder`	6		

ii Reeds verholpen bugs

Deze sectie geeft een zeer kort overzicht van enkele verholpen bugs uit de implementatie van SCALA-AM.

Bug #1

Het resultaat `equal?`-primitieve bij het vergelijken van twee lijsten was soms `true` in plaats van `false`.

Bug #2

De parser van het framework zou commentaartekst die zich over meerdere lijnen uitstrekte verkeerd parsen.

Bibliografie

[1] ABELSON, H., ADAMS IV, N.I., BARTLEY, D.H., BROOKS, G., DYBVIG, R.K., FRIEDMAN, D.P., HALSTEAD, R., HANSON, C., HAYES, C.T., KOHLBECKER, E., OXLEY, D., PITMAN, K.M., ROZAS, G.J., STEELE JR., G.L., SUSSMAN, G.J., WAND, M., *Revised^5 Report on the Algorithmic Language Scheme*, 1998.

[2] *Algo & Data 1*, Programmacode, internet, geraadpleegd op 2017-03-29, (ftp://cw.vub.ac.be/pub/courses/curriculum/AlgoDat1/programmacode/).

[3] *An Introduction to Scheme and its Implementation*, internet, geraadpleegd op 2017-03-25, (ftp://ftp.cs.utexas.edu/pub/garbage/cs345/schintro-v13/schintro_toc.html).

[4] DANVY, O., *Putting Scheme to Work*, internet, geraadpleegd op 2017-04-11, (http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/scheme/code/fun/).

[5] DYBVIG, K., *Chez Scheme Version 7 User's Guide*, internet, geraadpleegd op 2017-03-25, (http://www.scheme.com/csug7/).

[6] FEELEY, M., *Gambit Scheme*, internet, geraadpleegd op 2016-11-18, (http://github.com/gambit/gambit).

[7] *Holidays related to Easter*, internet, geraadpleegd op 2017-03-25, (http://rosettacode.org/wiki/Holidays_related_to_Easter#Scheme).

[8] MIT/GNU Scheme, *Quoting*, internet, (https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Quoting.html).

[9] *Oefeningenbundel*, internet, geraadpleegd op 2017-04-12, (soft.vub.ac.be/SCPI).

[10] *Principles of Object-Oriented Programming Languages*, internet, geraadpleegd op 2016-12-06, (http://pointcarre.vub.ac.be/index.php?application=weblcms&go=course_viewer&course=2338).

[11] *Roots of a quadratic function*, internet, geraadpleegd op 2017-04-13, (http://rosettacode.org/wiki/Roots_of_a_quadratic_function).

[12] *Sigscheme*, A R5RS Scheme interpreter for embedded use, internet, geraadpleegd op 2017-04-11, (https://github.com/uim/sigscheme/tree/master/bench).

[13] STIÉVENART, Q., *(Abstract) Abstract Machine Experiments using Scala*, internet, (http://github.com/acieroid/scala-am).

[14] STIÉVENART, Q., NICOLAY, J., DE MEUTER, W., DE ROOVER, C., *Building a Modular Static Analysis Framework in Scala (Tool Paper)*, In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, 105 – 109, ACM, New York, New York, VS, 2016.

[15] STIÉVENART, Q., VANDERCAMMEN, M., DE MEUTER, W., DE ROOVER, C., *Scala-AM: A Modular Static Analysis Framework*, In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 85 – 90, IEEE, Raleigh, Noord-Carolina, VS, 2016.

[16] VAN DER PLAS, J., *(Abstract) Abstract Machine Experiments using Scala*, internet, (http://github.com/jevdplas/scala-am).

[17] VAN DER PLAS, J., *Bachelor onderzoeksstage*, SCALA-AM, een framework voor statische codeanalyse, 2016.