Abstract state machines

Bachelorproef voorgedragen tot het behalen van de graad van bachelor in de informatica

Maarten Vangeneugden
Promotor: Prof. dr. Jan Van den Bussche

2018-2019

Inhoudsopgave

1 Voorwoord

Binnen het vakgebied van de informatica wordt er constant gezocht naar manieren om op een degelijke wijze programma's te ontwikkelen. Die programma's

moeten voldoen aan strenge eisen; correctheid, leesbaarheid, onderhoudsvriendelijkheid, en als het kan moeten ze ook nog snel genoeg zijn.

Deze programma's worden geschreven in zogenaamde "programmeertalen", een taal met een bepaalde grammatica waarin programmeurs hun algoritme kunnen uitschrijven zodat een computer weet wat het programma moet doen.

Deze programmeertalen zijn allemaal in meer of mindere mate gebaseerd op een berekeningsmodel. Dit betekent dat de programmeertaal een bepaalde formele vorm aanhoudt waarbinnen de berekeningen uitgevoerd zullen worden. Het Turingmodel is een veelgebruikt voorbeeld hiervan. Talen die dit model gebruiken, zullen ook de programmeur aanzetten hun algoritme in dit model uit te drukken.

De vraag is echter of algoritmen wel zo goed binnen die berekeningsmodellen passen. Misschien zijn er betere manieren om ze uit te drukken dan bv. in een strikt procedurale vorm.

Met het concept van Abstract state machines (ASM's) wordt getracht tegemoet te komen aan de implementatie van algoritmen naar goede software. Het betreft een berekeningsmodel waarin het idee van älgoritme"centraal staat, met als doel om algoritmes zó te formaliseren dat ze voor veel meer dan enkel software kunnen bepaald worden.

Deze bachelorthesis begeleidt de lezer in het begrijpen van de *abstract state machine*, en is drieledig:
Er wordt begonnen met een inleidend gedeelte, waarin de geschiedenis en situering van ASM's wordt toegelicht, alsook de formalisering van algoritmen voor de informatica. Daarna wordt de ASM zelf beschreven: De toepassingsvelden, de elementen, en eventuele uitbreidingen. Tot slot wordt er geraakt aan een programmeertaal die het ASM-model volgt, waarbij de theorie in de praktijk wordt omgezet.

In de tekst worden regelmatig vergelijkingen gedaan met programmeertalen die een pak meer bekendheid genieten onder informatici, om te trachten zo snel mogelijk tot een goed begrip van een ASM te komen voor de lezer.

ASM's zijn krachtige modellen, en zijn een prachtvoorbeeld van welke materie de informatica tracht te bestuderen. Het is dan ook spijtig dat er niet veel aandacht aan geschonken wordt; voor deze bachelorproef had ik er zelfs nog nooit van gehoord.
De beschikbare bronnen zijn dan ook relatief schaars. Ik heb daarom met volle overtuiging geprobeerd een zo duidelijk mogelijk beeld te scheppen. Zeker omdat haast alle eerdere werken in het Engels zijn geschreven, heb ik gekozen om deze thesis in het Nederlands te schrijven, omdat ik geloof dat talenkennis geen barrière mag vormen om dit onderwerp te kunnen leren.

Ik hoop dat deze thesis een waardevolle bijdrage kan leveren aan de wetenschap van de informatica.

2 Geschiedenis van modellen

Alan Turing was met zijn Turingmachine de eerste die een wiskundig berekeningsmodel bedacht, en zo de bakermat legde voor de informatica als exacte wetenschap.

Sinds dat moment zijn informatici altijd op zoek geweest naar de grenzen van het berekenbare, en dat heeft geleid tot verbluffende resultaten.

> Ik ben meer dan dertig jaar radioloog. En dit is voor mij de meest opwindende vernieuwing in onze job.[14]
>
> *prof.dr. Paul M. Parizel over AI-analyse in de radiologie*

Echter, de focus van deze modellen heeft altijd gelegen op "berekenbare functies", en minder op "berekenbare algoritmen".

Een groot probleem van de huidige manier waarop software ontwikkeld wordt, is dat er maar zelden garanties worden gegeven over de werking ervan. Veel software komt dan ook met een vrijwaringsclausule in de licentie.
Het is (tot nu toe) ook altijd moeilijk geweest om garanties te geven; er zijn talloze factoren die de werking van het programma kunnen aantasten. De steeds groter wordende complexiteit van onze software mag hierbij niet onderschat worden.

Over de tijd heen zijn er verschillende methoden bedacht om toch een bepaalde graad van zekerheid af te kunnen dwingen, in de vorm van *test driven development*, *continuous integration*, ...

In de jaren '80 bedacht Yuri Gurevich het idee van een "evoluerende algebra". Hij stelde dat "elk algoritme, hoe abstract het ook is, kan stap voor stap beschreven worden middels een evoluerende algebra". Later maakte de term plaats voor "abstract state machine".

De ASM-thesis was vernieuwend in de zin dat de notie van algoritmische abstractie verbreedt werd tot algoritmen die niet met een computer kunnen worden berekend. Kookrecepten, reanimatie, ... zijn voorbeelden van algoritmen die wel met een ASM kunnen beschreven worden, maar niet met een Turingmachine, althans niet op een werkbaar abstractieniveau.

Dit doet geen afbraak aan het nut van eerdere modellen. De eenvoudigheid van een Turingmachine is op z'n minst opmerkelijk te noemen voor wat het kan simuleren. ASM's zijn gecompliceerder en er is dan ook geen praktisch gedachte-experiment om een ASM uit te beelden.

2.1 Het ontstaan van ASM's

Abstract state machines (of "Evoluerende algebra's" zoals ze origineel genoemd werden) zijn ontstaan omdat er een merkbaar verschil zit tussen berekeningsmodellen en specificatiemethoden [7, Introduction]. Dat is spijtig; veel toepassingsgebieden hebben duidelijk baat bij een combinatie van beide technieken. Er was ook nog geen manier om een algoritme te beschrijven op zo'n manier dat er geen programmacode nodig was. Aan de ene kant bestond er de specificatie ("Het algoritme geeft de snelste route terug"), maar dat vertelt nog niets over hoe het algoritme precies in zijn werk gaat. De thesis omtrent evoluerende algebra's ontstond om aan te tonen dat het wel degelijk mogelijk is om zo'n beschrijving te geven, aangenomen dat er een algebra werd opgesteld voor dat algoritme.

Het doel van de ontwikkeling is dus om een brug te slaan tussen die twee concepten. Dit zou het mogelijk moeten maken om met eenzelfde hulpmiddel zowel een specificatie als een werkend algoritme te beschrijven. Daarvoor diende er wel een manier te zijn om een systeem te hebben dat dicht bij de realiteit staat, maar ook dicht bij het algoritme zelf. ASM's zijn dus ontwikkeld om zo'n systeem te hebben, en bieden ook de benodigde onderdelen om zelf aan de slag te gaan. Als het mogelijk is om elk algoritme te beschrijven als een ASM, en

dit op elk abstractieniveau, dan is dat empirisch bewijs van de validiteit van de ASM-thesis [9].

3 Toepassingsvelden ASM's

Het belang van een nieuw concept aantonen is vitaal om het gebruik ervan te kunnen aanmoedigen; men moet overtuigd zijn van ASM's als een bruikbaar hulpmiddel in de problemen die een informaticus probeert op te lossen.

In dit hoofdstuk worden de toepassingen van ASM's tegen het licht gehouden, zowel de positieve als negatieve mogelijkheden.

3.1 Veelbelovende toepassingen

3.1.1 Bewijs van correctheid

Programma's die correct werken zijn in de meeste gevallen praktisch zeer wenselijk, en soms zelfs van levensbelang. Een model dat het makkelijk maakt om die correctheid dan aan te tonen is in die opzichten dan ook een dankbaar hulpmiddel.

In programmeertalen zoals Python, C, Kotlin, ... wordt er speciale code geschreven die dan als het ware de code van het programma test; er wordt input gegeven, en gecontroleerd op de verwachte output. Komt dit niet overeen met de gegeven output, dan duidt dat op een fout in het algoritme[1].

Met ASM's wordt de *trial-error*-manier van testen vervangen door het gebruik van wiskundige modellen om correctheid aan te tonen. Een wiskundig correctheidsbewijs is per definitie niet onderhevig aan mogelijke fouten in het bewijs zelf (lees: als het bewijs wiskundig klopt, dan is het ook met zekerheid correct). Het is voor veel mensen natuurlijk leuk als hun software juist werkt, en met ASM's is dat makkelijker te garanderen dan mogelijk met andere programmeertalen. Natuurlijk kan indien nodig nog altijd met *trial-error* gewerkt worden om te controleren of de ASM-specificatie ook correct is geïmplementeerd.

Dit kan verregaande voordelen opleveren, waaronder ook voordelen die voor gewone mensen duidelijk zichtbaar zijn: Het is hoogst ongebruikelijk dat nutsvoorzieningen zoals water of elektriciteit niet beschikbaar zijn (althans in België), of toch zeker niet zonder een waarschuwing lang op voorhand. Als het op digitaal verkeer aankomt zijn de storingen echter legio [13] [16] [6] [15] [5] [10]. De garantie die het gebruik van ASM's kan bieden voor de beschikbaarheid van systemen kan ongetwijfeld een positieve economische impact bieden.

3.1.2 De ASM-methode

Alhoewel ASM's wiskundig gezien krachtig genoeg zijn om als basis voor programmeertalen te dienen, is dit nogal beperkend voor de mogelijkheden die het biedt.

De ASM-methode is een werkmethode waarbij de brug gemaakt wordt tussen de formalisering en begrip van het op te lossen probleem enerzijds, en het schrijven en testen van de reële implementatie van het algoritme anderzijds.

[1] Hierbij dient opgemerkt te worden dat het hier over de *implementatie* van het algoritme gaat. Het is goed mogelijk dat het idee van het algoritme correct is, maar dat de manier waarop het geschreven is niet in overeenstemming is met dat idee.

Hierbij zal de informaticus trachten om het probleem van de klant te abstraheren tot een (verzameling) algoritmen, die dan als een ASM kunnen worden voorgesteld.
Daarna kunnen de gemaakte ASM's dienen als voorbeeld voor de te schrijven programma's, waarbij de ASM's dan ook gebruikt worden om de correctheid aan te tonen.

Met deze methode worden ASM's niet aangewend als het programmeerparadigma op zich maar als een complementair gedeelte van het volledige programma. Ze vervangen programmeertalen niet, maar vormen een uitbreiding van de gereedschapskist van de informaticus.

3.1.3 Inleiding tot bewijsvoering

Van studenten-informatici wordt verwacht dat ze leren algoritmen te bewijzen op hun correctheid. Op dit moment wordt dat vooral gedaan met propositieleer, maar deze methode is nogal omslachtig.

Het kunnen opstellen van een ASM zou een waardevolle aanwinst zijn voor deze vaardigheden. Zo kunnen ze ook gemakkelijker leren communiceren met latere klanten over hun problemen.

3.1.4 Grammatica van programmeertalen

ASM's hebben hun nut al vele keren bewezen, ondanks de relatieve onbekendheid. De programmeertaal Prolog is zo'n voorbeeld, want de hele taal is wiskundig gedefinieerd met behulp van ASM's. ASM's zijn ook voor dit nut aangewend voor Java en C.

3.1.5 Hardwarebeschrijvingen

Er werd eerder al gezegd dat ASM's zich niet beperken tot berekenbare functies. Het model van een ASM leent zich ook goed voor ingenieurs die hardware willen beschrijven. Dit laat toe om op een gemakkelijke manier analyses uit te voeren over de interne werking van de systemen.

3.2 Minder geschikte toepassingen

3.2.1 Inleidend programmeren

ASM's zijn geen goede keuze om studenten te laten kennismaken met programmeren, en zouden hiervoor ook beter niet gebruikt worden.
Zeer problematisch is het gebrek aan programmeertalen die hiervoor geschikt zijn.

Talen zoals Scheme en Python abstraheren een hoop processen om het aanleren van programmeren gemakkelijker te maken, in tegenstelling tot talen zoals C: Denk aan manueel geheugen alloceren, pointers, foutmeldingen, I/O, ... Deze technieken[2] zijn hinderlijk als men een student wilt leren om algoritmen procedureel/functioneel op te schrijven.

[2]Dit wilt niet zeggen dat informatici in wording deze technieken niet moeten beheersen, maar binnen bepaalde vakken zal dit meer hinderlijk dan leerrijk bevonden worden. Als het dan mogelijk is om hier omheen te werken, dan mag hier dankbaar gebruik van worden gemaakt.

Voor ASM's bestaan er op dit moment geen talen die op dergelijke wijze het leren kunnen vereenvoudigen. Om volgens dat model dan ook programma's te schrijven moet de student dus ook een hoop code schrijven die strikt genomen niets toevoegt aan wat er moet aangeleerd worden, waarop later wat dieper wordt ingegaan.

3.2.2 Snelle/Kleine oplossingen

Soms moet er maar een klein programmaatje gemaakt worden, of rust er een zware tijdsdruk op het project.
Een ASM opstellen vereist aandacht en denkwerk, maar waarschijnlijk staat die voorbereiding soms niet in verhouding tot het uiteindelijke resultaat. Misschien is het programma zelfs zo klein dat de correctheid op het oog te verzekeren is. In zo'n geval zie ik AMS's niet als een handig hulpmiddel, maar meer een onnodige last.

4 Definitie van de abstract state machine

4.1 De fundamenten

Net zoals in de fysica het standaardmodel wordt gedefinieerd door fundamentele deeltjes, wordt een abstract state machine gedefinieerd door fundamentele elementen:

- Vocabulaire
- Functie
- Regel
- Toestand
- Invariant

Deze onderdelen volstaan om elk algoritme te modelleren, maar ook om de correctheid ervan te bewijzen. Verder in de thesis worden deze elementen elk apart in detail beschouwd. Nadien wordt bij de bespreking van ASMeta het verband tussen de programmeertaal en de theorie aangebracht.

4.2 Een (evoluerende) algebra?

Een ASM vertoont veel eigenschappen die we terugvinden in wiskundige algebraïsche structuren; een bepaalde structuur, waarop een verzameling van operaties kan worden uitgevoerd. Er zijn echter ook uitbreidingen: Zo kan het resultaat van die operaties in een ASM veranderen naarmate het programma vordert.

Omwille van die gelijkenis wordt "ASM" in oudere werken dan ook "evoluerende algebra" genoemd.

De lezer kan zelf beslissen hoe per[3] het idee benoemt; in een wiskundige

[3]Het Nederlands kent geen persoonlijk voornaamwoord dat naar mensen verwijst zonder geslachtsaanname. Daarom wordt doorheen deze thesis het woord "per" hiervoor gebruikt, afkorting van "persoon". Dit woord past goed in de Nederlandse taal, en heb ik ontleend van het gebruik ervan in het boek Woman on the Edge of Time van Marge Piercy, waar het dezelfde functie vervult voor het Engels.

context kan “ASM” de link met de algebra onnodig verbergen, terwijl in de informatica “evoluerende algebra” misschien niet direct laat vermoeden dat het een volledig berekeningsmodel betreft.

4.3 Voorbeeld: Webserver

Om te illustreren hoe we met een ASM praktische toepassingen kunnen ontwerpen, zullen we tijdens het uitleggen van de elementen een webserver opbouwen, geïnspireerd op een opdracht omtrent websockets uit het opleidingsonderdeel Computernetwerken, waarin we omgaan met *clients* en communicatie behandelen.

5 Abstractieniveaus

Als we aan de abstractie van een probleem denken, dan proberen we om zoveel mogelijk niet-gerelateerde bewerkingen uit het probleem weg te filteren.

Programmeertalen bestaan er in alle vormen en kleuren. Hun verschillen worden vaak op een reeks bekende spectra geplaatst, bekende voorbeelden zijn:

Dominante paradigma('s) Object-geöriënteerd, logisch, functioneel, ...

Typesysteem Statisch, dynamisch, inferentie, zwak, sterk, gradueel, ...

Platform Gecompileerd naar machinecode, een *virtual machine*,

geïnterpreteerd, gescript, ...

Symboolbereik Lexicaal, globaal, dynamisch, ...

Deze geven oppervlakkige informatie over de programmeertaal, maar vertellen helemaal niets over het abstractieniveau waarop de code geschreven dient te worden.
Toch zien we binnen Turingtalen sterke verschillen in abstractie van het algoritme. In C moet de programmeur constant manueel het geheugen alloceren indien het programma dat vereist, Strings bestaan eigenlijk niet, en om tekst naar het scherm printen moet de programmeur aangeven wat het type van de af te drukken variabele is.
Vergelijk met Python: Geheugenallocatie wordt geabstraheerd; de programmeur kan het algoritme gewoon beschrijven alsof dat niet nodig is. Daarnaast biedt Python ook een hoop structuren aan die in een lager abstractieniveau niet bestaan.

Deze abstractieniveaus hebben wel allemaal één ding gemeen: Ze abstraheren enkel processen op computers, maar niet de processen in de realiteit.

5.1 Abstractie van algoritmen

Abstractie hoeft echter niet beperkt te zijn tot de ideeën die op computers bestaan.

Neem als voorbeeld een algoritme dat een bankautomaat beschrijft. Welk abstractieniveau is hiervoor het best geschikt?
Het is zeer belangrijk om te weten dat een bankkaart in de automaat aanwezig

is. Dat moet dus door het algoritme gecontroleerd worden.
Echter, de kaart moet een leesbare chip hebben. De manier waarop die leesbaarheid gecontroleerd wordt is echter niet nuttig voor het algoritme zelf. Dat proces valt dus buiten de gekozen abstractie.

Het probleem dat zich nu stelt met andere berekeningsmodellen, is dat men hier tóch het controleproces in rekening zal moeten brengen, ook al is dat proces voor ons niet van belang.

Het valt nu op dat ASM's in zekere zin *krachtiger* zijn dan Turingmachines: Hiermee kunnen abstractieniveaus (ANs) bereikt worden die met andere berekeningsmodellen niet mogelijk zijn. Met andere woorden: *Veel algoritmen kunnen beschreven worden met een equivalente ASM, waaronder ook algoritmen die niet op een computer geïmplementeerd kunnen worden.*

5.2 Webserver

Vooraleer we regels en functies kunnen gaan schrijven is het dus noodzakelijk om eerst te kijken **wat** onze webserver nu eigenlijk allemaal moet kunnen, en welke onderdelen ervan niet echt relevant zijn aan de definitie ervan.

Dit gaat het makkelijkst door gewoon al eens op te schrijven wat het moet kunnen:

- Een aanvraag (`REQ`) ontvangen van een potentiële *client*, een verbinding openen
- Een eerder geopende verbinding ook afsluiten
- Tijdens de verbinding: communiceren met de *client*, informatie uitwisselen
- Communiceren met andere software die de gegeven informatie kan interpreteren
- Bij meerdere gebruikers: bepalen wie eerst "geholpen" wordt

Hiervoor zullen we dus de gepaste regels en functies moeten beschrijven.

Er zijn echter nog een hoop andere dingen die een webserver moet doen, maar die voor onze abstractie niet wensenlijk zijn. Denk aan complexere opdrachten zoals:

- *Network Address Translation* voor IPv4-adressen
- DDoS-detectie
- SSL-versleuteling, encryptie en decryptie
- Andere aanvallen door crackers verhinderen
- Transmissiefouten opvangen en corrigeren
- TCP *packet assembly*
- ...

Deze lijst beschrijft maar een klein aantal voor de hand liggende concepten, maar denk ook aan de kleinere problemen: Een *client* die tijdens het verzenden de verbinding verliest, kan een onvolledig bericht verstuurd hebben, maar misschien was ie net klaar. Opnieuw, voor ons abstractieniveau laten we deze problemen buiten beschouwing.

Dat wil niet zeggen dat we gewoon maar aannemen dat deze functionaliteit niet nodig is voor de webserver, maar dat ons abstractieniveau dicteert dat we zullen aannemen dat deze functionaliteit er al is. We gaan dus in ons ASM geen code schrijven om te differentiëren tussen IPv4 en IPv6, omdat het voor ons niet uitmaakt wat de *client* gebruikt.[4]

Vergelijk het een beetje met een kookrecept: Er zal altijd wel worden beschreven hoe een ajuin gesneden moet worden (gehalveerd, fijn, grof, ...), maar er wordt nooit vermeld dat men eerst een mes moet vastnemen. Er wordt niet gesuggereerd dat dat niet nodig is, maar het recept gaat ervan uit dat de lezer dat weet, om zo het beschrijven van het recept niet nodeloos te bemoeilijken.

6 Functie

Herinner dat een ASM draait rond het modelleren van algoritmen. Het spreekt dan vanzelf dat er ook een uitgebreide notie van functies bestaat.

Functies in ASM's zijn enorm veelzijdig; ze worden gebruikt als variabelen, als subroutines, als communicatiepoorten met "de buitenwereld", ... In dit hoofdstuk worden dan ook de verschillende vormen waarin ze voorkomen opgenoemd.

6.1 Eigenschappen van de functie

Een functie in een ASM heeft steeds een vooraf vastgelegde, eindige ariteit, die niet kan wijzigen.
Een functie geeft steeds één waarde terug, zonder uitzondering. Wel kan een functie een tuple teruggeven, die dan uit meerdere waarden bestaat, te vergelijken met hoe functies in Python ook meerdere waarden kunnen teruggeven.[5]

Er zijn geen strikt vastgelegde regels over de notatie van functies. Over het algemeen worden functies met de prefixnotatie geschreven, bijvoorbeeld: `ExampleFunction(x,y)`
Voor relationele functies is het, afhankelijk van de functienaam, ook gebruikelijk om de functies infix te schrijven. Goede voorbeelden zijn $\in$ en $=$.

Zeker in regels bij het gebruik van *guards* is het mogelijk om meer in zinnen te schrijven dan in een strikt vastgelegde grammatica. `if x is a node` is bijvoorbeeld perfect mogelijk, en zou kunnen slaan op een relationele functie `node(x)`.

Functies zijn steeds **atomair**; de programmeur mag aannemen dat een functie steeds volledig zal worden uitgevoerd.

[4] In de praktijk kunnen we natuurlijk niet gewoon tegen een computer zeggen dat we dat "geabstraheerd" hebben en verwachten dat het wel zal weten wat we bedoelen. De stijl van ons programma zou niet wijzigen, maar we zouden wel ASM's op lagere abstractieniveaus moeten beschrijven, waarmee we dan binnen de programmeertaal communiceren.

[5] In tegenstelling tot wat programmeurs vaak denken, kunnen Python(3)-functies slechts 1 waarde teruggeven. Echter, de manier waarop dat gebeurt is *syntactic sugar* voor het in- en uitpakken van een tuple.

Formeel worden functies steeds als **totaal** beschouwd, er is dus een output gedefinieerd voor elke mogelijke input. De nuance vereist te vermelden dat deze totaliteit wordt waargemaakt dankzij `undef`, die wordt gegeven als er geen andere term mogelijk is.

6.2 Statische functie

Veel mensen denken bij een (wiskundige) functie aan een invariante entiteit; voor een gegeven input krijgt men altijd dezelfde output.
Deze stelling gaat niet op voor alle functies zoals die in een ASM voorkomen, maar omdat deze soort functies een speciaal geval vormen, worden ze gecategoriseerd als "statische functies".

Deze functies zullen tijdens de uitvoering van een ASM niet veranderen. Een bekend voorbeeld zijn de goniometrische functies; $cos(\pi)$ zal altijd 1 geven.

6.3 Dynamische functies

Waar statische functies zijn, zijn er ook dynamische, en deze zijn een pak interessanter. Bij deze functies is het namelijk mogelijk dat hun output wél varieert doorheen het programma. De manier waarop dat gebeurt bepaalt ook hun onderverdeling.

6.3.1 Externe functies

Vaker dan niet zullen algoritmes die in de praktijk gebruikt worden nood hebben aan informatie die niet in de ASM voorkomt (de zogenaamde "niet-gesloten" algoritmen). Statische en dynamische functies volstaan hier niet voor.

Externe functies bieden hier uitkomst. Deze functies maken interactie met de omgeving mogelijk, waardoor er input kan verkregen worden van de omgeving, en output kan gegeven worden.

Wat wel belangrijk is, is dat een externe functie voor eenzelfde stap **dezelfde uitwerking** moet hebben [7, External Functions, p. 13]. Men zou zelfs kunnen stellen dat een externe functie vervalt tot een dynamische functie binnen een stap. Tijdens het programmeren zal dan ook moeten worden afgedwongen dat de programmeur niet meerdere keren in 1 stap dezelfde functie uitvoert.
Dit is nodig om de atomaire eigenschap van functies tegemoet te komen.

Dit principe van een categorie functies benoemen voor externe acties is nauw verwant aan het functioneel paradigma:
Bij functioneel programmeren wordt er gesteld dat er geen *toestand* is; alle functies bestaan in een theoretisch gefixeerde ruimte. In de praktijk (en dus in de programmeertalen) heeft dit weinig nut als er geen verband is met de eigen wereld. Dus voorzien functionele programmeertalen speciale functies die *impure functies* genoemd worden; deze zijn in staat om te interageren buiten de functie, en kunnen dus globale variabelen aanpassen, communiceren met een *web socket*, input van de gebruiker vragen, ...
In die zin hebben impure functies dus een belangrijke eigenschap gemeen met externe functies.[6]

[6] Voor de volledigheid: Van pure functies wordt ook verwacht dat ze voor een bepaalde input steeds dezelfde output geven. Impure functies kunnen dus ook in een ASM bestaan als een dynamische functie.

Het complement van de externe functies worden de interne functies genoemd. Stel S een toestand binnen een programma, dan is S^- die toestand met de vocabulaire reduceerd tot de interne functies.

6.3.2 Interne en gedeelde functies

Het merendeel van de functies in een ASM zullen echter van het interne type zijn, wat betekent dat deze enkel binnen het algoritme zelf kunnen worden aangepast. Als het algoritme het vereist, dan kan een ASM ook een gedeelde functie hebben. Deze functies zijn zowel intern als extern bereikbaar.

6.4 Relaties

Om relaties uit te drukken tussen waarden, worden in ASM's ook functies gebruikt.
Een relationele functie heeft als speciale eigenschap dat het slechts twee waarden kan teruggeven, `true` of `false`. Deze functies dienen dus om aan te geven dat er een relatie ís, niet wat deze relatie inhoudt. Zo is een functie `edgeBetween(Node, Node)` relationeel, maar `edgeWeight(Node, Node)` niet, ook al is er duidelijk een relatie mee gemoeid.

6.5 Termen

Zoals eerder vermeld zijn functies in de ASM's zeer abstracte en polyvalente elementen.
Om binnen een ASM het concept van een variabele te kunnen hebben, wordt dan ook gebruik gemaakt van functies. Deze functies noemen *termen*, en moeten aan de volgende voorwaarde voldoen:

Een functie is een term $\Leftrightarrow$ al haar argumenten termen zijn.

Uit deze voorwaarde volgt dat een functie met ariteit nul triviaal ook een term is [9, Derivation of Sequential ASMs, p. 13].

6.6 De ASM als functie

Alle ASM's zijn equivalent aan een bepaalde functie, en voor elke functie bestaat een equivalent ASM. Deze equivalentie-eigenschap is zeer belangrijk om het concept van abstractieniveaus werkbaar te maken. Het is de bedoeling dat informatici met *abstract state machines* beginnen zeer kleine algoritmes te modelleren, en die dan te hergebruiken in hogere niveaus als functies om steeds complexere problemen te kunnen modelleren. Op die manier kan er een netwerk van ASM's ontwikkeld worden die correcter en sneller kunnen werken dan de huidige oplossingen.

6.7 Webserver

We gaan trachten de verschillende soorten functies volledig te benutten.

De verbinding tussen een *client* en de server is een goed voorbeeld van een relatie. Stel `isConnected(client)`, dewelke `true` of `false` teruggeeft als er al dan geen verbinding is.

Onze webserver zal ook achterliggend communiceren met andere programma's die, afhankelijk van wat een *client* geeft, wat andere data moet teruggeven. Dit

is een duidelijk voorbeeld van **twee** externe functies: `clientChannel(data)` en `softwareChannel(data)`. Deze functies werken in beide richtingen: Ze kunnen data teruggeven, maar kunnen ook vanuit onze webserver voorzien worden van data. Dat past ook mooi binnen het abstractieniveau dat we hebben gekozen.

We zouden ook een manier moeten hebben om te vragen of er zich nieuwe *clients* aandienen. Ook dit zou dan een externe functie zijn: `newClients(client)`. Deze functie zal enkel echte *clients* teruggeven, en niet blindelings elke verbinding als valide beschouwen. Herinner dat we beveiliging buiten ons abstractieniveau laten.

Hoe gaan we om met verbroken verbindingen? Het kan voorvallen dat een *client* onbedoeld de verbinding verliest, maar van het standpunt van de server maakt dat geen verschil. Alle *clients* die hun verbinding verloren kunnen worden opgevraagd via `lostClients(client)`. We gaan dit later nodig hebben om de toestand van de webserver te kunnen aanpassen.

Om de ASM niet onnodig te compliceren, behandelen we slechts één *client* per *run*. De functie `activeClient` geeft aan welke *client* op dit moment geholpen wordt.

De webserver bezit nu de benodigde functies om de opdrachten voor ons abstractieniveau uit te voeren. De volgende taak is om die functies ook in te zetten binnen het ASM, en daarmee zullen de toestanden en regels onontbeerlijk zijn.

7 De regel

Één van de fundamentele elementen van een ASM is de *rule*, of *regel* in het Nederlands.
Een regel beschrijft veranderingen die optreden bij een gegeven toestand van het programma, en vervult dus een opdracht bij de overgang van een toestand A naar een (al dan niet verschillende) toestand B. Hierdoor worden ze soms ook wel *transitieregels* genoemd.

Het gebruik van een regel vereist **toepasbaarheid** bij een toestand. Toepasbaarheid is gedefinieerd als volgt: [8, Informal Semantics, p. 11]

> A rule R and an expanded state A are appropriate for each other if Voc(R) $\subseteq$ Voc(A) and FreeVar(R) $\subseteq$ Var(A).

Informeel betekent dit dat de toestand waarop de regel wordt toegepast alle vereiste variabelen en functies moet hebben die de regel nodig heeft.

Regels zijn op zich ook programma's: Met een bepaalde input gaan ze aan de slag, en geven ze een bepaalde output terug. Deze beschrijving is echter verwarrend, omdat regels altijd worden gezien als onderdeel van het grotere programma, de eigenlijke ASM (dat op zich ook als regel kan worden beschouwd). Daarom wordt in deze thesis met "programma" de instinctieve betekenis bedoeld, tenzij anders vermeld.

Elke regel in een ASM beschrijft één stap binnen het algoritme dat de ASM modelleert. Na het uitvoeren van een regel tijdens een toestand S zal er steeds een toestand S' op volgen, die ook deel uitmaakt van diezelfde ASM. Deze programmeerstijl dwingt de programmeur om hetzelfde AN aan te

houden. En dat AN zorgt ervoor dat het algoritme correct en overzichtelijk blijft.

Merk op dat regels géén deel uitmaken van de vocabulaire! Ze maken deel uit van de algoritmische beschrijving en gebruiken de namen wel, maar buiten dat zijn ze een verzameling op zich. Toch geldt (net zoals voor de vocabulaire) dat de regels vastgelegd zijn, en ze kunnen niet worden gecreëerd of verwijderd.

7.1 Soorten regels

Alle regels zijn fundamenteel van dezelfde vorm: $f(x_1) := x_0$, waarbij de regel een bepaalde toestand omzet naar een andere toestand. De mogelijke regels zijn onderverdeeld in bepaalde categorieën. Zij worden hier toegelicht, samen met hun wiskundige beschrijving.

7.1.1 Skip

Deze regel doet niets. Deze regel bestaat om de definitie van andere regels gemakkelijker te stellen, zoals een voorwaardelijke regel; als de programmeur een verandering enkel wilt doorvoeren als aan voorwaarde p voldaan is, dan zal de regel in niet-voorkomend geval een *skip* zijn.

7.1.2 Update

De update-regel beschrijft een verandering binnen de toestand van het programma.

Update-regels worden genoteerd als $f(x, y, ..., z) := t$, met t de gegeven term voor de termen x, y, ..., z. Als f een relationele functie is, dan zal t `true` of `false` zijn. Het is aan de programmeur om te werken met een een ander type als dit beter de semantiek van de relatie beschrijft (denk aan een relatie tussen twee knopen die het gewicht van een boog teruggeeft).

7.1.3 Voorwaardelijk

Een voorwaardelijke regel betreft een if-else-statement waarin de uitkomst afhankelijk is van de toestand waarop deze regel wordt toegepast. Concreet wordt dit genoteerd als:

```
if <expressie> then
<regel>
elseif <expressie> then
<regel>
...
else
<regel>
endif
```

De `if` is verplicht, maar `elseif` en `else` kunnen weggelaten worden als daarvoor geen regel moet worden uitgevoerd. In dat geval kan er worden gesteld dat het complement van de if-expressie een *skip*-regel tot gevolg heeft.

Een voorwaardelijke regel kan ook worden opgevat als twee regels: De pre- en postcondities van de voorwaardelijke regel zijn de unie van die van de regels

waaruit de regel bestaat; er moet namelijk maar aan één van beide regels voldaan worden. Het volgt dan triviaal dat de beslissende voorwaarde een *guard* vormt zodat de uitgevoerde regel toepasbaar blijft.

7.1.4 Blok

Een verzameling van regels die worden uitgevoerd is zelf ook een regel. De blokregel begint door het sleutelwoord `do in-parallel` te geven, waarna elke uit te voeren regel wordt geschreven, gevolgd door een afsluitende `enddo`. `in-parallel` mag eventueel weggelaten worden.

Merk op dat een verzameling geen volgorde kent; dit impliceert dus dat de regels binnen een blokregel tegelijkertijd kunnen worden uitgevoerd. In de praktijk komt dit erop neer dat een compiler de regels probleemloos op verschillende *threads* kan zetten, wat de uitvoeringssnelheid zeker ten goede zal komen. Dit is een eigenschap van ASM's die onmogelijk is voor procedurele programmeertalen.

Moest een sequentie toch nodig zijn, dan dient `in-parallel` vervangen te worden door `in-sequence`.

Strijdige regels Een regel kan in strijd zijn met een andere regel binnen een blok, [8, Informal Semantics, p. 11] maar dan verliest de blokregel haar toepasbaarheid voor alle mogelijke ASM's. Dit kan als volgt bewezen worden:

Bewijs. Stel B een blokregel, een verzameling van regels. Stel voor r_1 geldt dat $x \in \mathbb{N}$ en voor r_2 $x \notin \mathbb{N}$, met $r_1, r_2 \in B$

De doorsnede van de precondities van r_1 en r_2 geeft dat $x \notin \mathbb{N} \wedge x \in \mathbb{N}$.

Er zijn nu twee mogelijkheden van toestanden:

- De toestand bevat x :: Dan is er geen toestand die kan voldoen aan de preconditie die wordt opgelegd aan x.
- De toestand bevat geen x :: Dan is de vocabulaire van B geen deelverzameling van die van de toestand, waardoor B niet toepasbaar is.

In beide gevallen is B niet toepasbaar. Er bestaat dus geen toestand waarin strijdige regels mogelijk zijn. □

Merk op dat het verbod op strijdigheid nondeterminisme niet in de weg hoeft te staan. Zie nondeterminisme als uitbreiding over dit onderwerp.

Sequentieel of parallel Blokregels worden parallel uitgevoerd. Als de volgorde van uitvoering er toe doet, kan dit ook aangegeven worden met `do in-sequence`. Als er enkel `do` genoteerd staat, wordt de paralle (standaard) versie gebruikt.

7.1.5 *For all*

De *for all*-regel lijkt sterk op wat gekend is als een *loop* in andere talen, maar werkt iets anders.

Dit is een speciale vorm van een blokregel, in die zin dat er ook meerdere regels worden uitgevoerd. Het verschil zit in het voorwaardelijke en variabele aspect, zoals duidelijk blijkt uit de notatie:

```
do for all x, y with <logische expressies>
<regel>
<regel>
```

```
...
<regel>
enddo
```

De variabelen `x` en `y` worden bepaald op basis van de logische expressies, zoals verder zal duidelijk worden in het voorbeeld van de webserver.

Merk op dat dit, ondanks dat het parallel lijkt, perfect binnen de notie van sequentiële ASM's past. Het algoritme kan misschien niet-sequentieel zijn, maar zolang dat het niet oneindig blijft doorgaan is dit geen belet om de ASM als sequentieel te beschouwen [9, Sequential algorithms].

7.2 Webserver

Dankzij de regels gaan we de eerder gedeclareerde functies aan elkaar koppelen.

Beginnen bij het begin: Behandelen van nieuwe aanvragen:

```
rule new-connections:
do for all client where newClients(client) = true
isConnected(client) := true
enddo
```

En dus ook de verloren verbindingen:

```
rule lost-connections:
do for all client where lostClients(client) = true
isConnected(client) := false
enddo
```

Voor het bedienen van de *clients* is er ook een regel nodig:[7]

```
rule determine-active-client:
choose client from isConnected where
isConnected(client) = true:
activeClient := client
endchoose
```

Tot slot hebben we ook een regel nodig die de webserver ook toelaat iets te *serven*:

```
rule communicate:
   do in-sequence
   softwareChannel(clientChannel)
   clientChannel(softwareChannel)
   activeClient := undef
   enddo
```

De sequentie is hier belangrijk: De client moet eerst informatie geven vooraleer de software daarop kan antwoorden. Daarna kunnen we ook pas instellen dat de *client* niet meer als actief beschouwd moet worden.

[7] De `choose`-regel is nog niet beschreven, maar zal straks voorkomen bij de uitbreidingen op ASM's.

8 Invarianten

Bij regels wordt er altijd vooropgesteld dat de toestand waarin ze worden toegepast, ook aan bepaalde voorwaarden moet voldoen.

Echter, om álle beperkingen af te dwingen zou redundant zijn.
Bijvoorbeeld: in een sorteeralgoritme is het logisch dat de gesorteerde lijst nooit langer kan zijn dan de ongesorteerde lijst.
Om dit te stellen voor elke regel zou maar verwarrend zijn. Daarnaast kan het perfect zijn dat een regel correct kan werken zonder die voorwaarden.

In ASM's is het daarom mogelijk om invarianten te bepalen op variabelen in de toestand. Deze invarianten stellen limieten die in het gehele programma geldig dienen te zijn.

Dit heeft twee handige voordelen: Voor wiskundige bewijsvoering laten invarianten toe makkelijker stellingen te maken. Anderzijds is het een garantie voor de programmeur; als een invariant overtreden wordt, dan klopt het programma ook niet, en wordt de programmeur ook direct op de hoogte gebracht van waar de fout optrad.

8.1 Beschrijving via toestanden

De invariant is geen fundamenteel element van de ASM, maar een idee dat met fundamentele elementen kan worden beschreven. In de geraadpleegde bronnen worden invarianten ook niet beschouwd als inherent aan de definitie van een ASM. Niettegenstaande heb ik toch besloten het zo te benoemen.

Rekening houdend met de parcimonie, kan dit beschouwd worden als een fundamenteel element van een ASM; het biedt pragmatisch gezien een aanzienlijke vereenvoudiging aan van algoritmes zonder afbreuk te doen aan correctheid, wat toch het kerndoel is van een berekeningsmodel.

Programmeertalen zijn vrij om met bijvoorbeeld *syntactic sugar* de indruk te wekken dat het om een fundamenteel element gaat.

Stel een ASM A met:

- Errortoestand E
- De variabelen v_1, v_2, ... v_n, + errorvariabele v_e en andere niet nader genoemde toestanden.

Voor alle regels van A geldt als preconditie: `v_e = false`. Deze preconditie is `true` voor alle toestanden, behalve E. Stel een regel t die een aantal variabelen controleert. Als een variabele niet voldoet aan de voorwaarden die in t gecontroleerd worden, dan `v_e := true`. In E kan de ASM geen updates meer doorvoeren, dus loopt het programma vast omwille van een ongeldige invariant.

8.2 Webserver

De invarianten van onze server zijn vooral van praktische aard. Zo kunnen we de waarden die de functies hebben beter specifiëren met enkele invarianten:

- $\nexists$ `x (newClients(x)` $\wedge$ `lostClients(x))`
- $\nexists$ `x (lostClients(x)` $\wedge$ `¬isConnected(x))`

- ∄ x (newClients(x) ∧ isConnected(x))
- lostClients(activeClient) = false
- newClients(activeClient) = false
- isConnected(activeClient) = true

De invarianten lijken misschien triviaal, maar het helpt om bewijskracht te leveren aan een formele modellering. Deze webserver dient nu slechts ter illustratie, maar voor grotere programma's kunnen de invarianten veel complexer worden, en dus ook een stuk minder intuïtief lijken. Zeker dan blijkt hun nut in het opstellen van wiskundige bewijzen.

9 De toestand

Het spreekt voor zich dat in een Abstract *state* machine het begrip *state* (oftewel *toestand* in het Nederlands) van groot belang is.

Een toestand S van een ASM A is een bepaalde toestand waarin A zich kan bevinden. S wordt uniek bepaald van de andere toestand door aan de vocabulaire V in S specifieke waarden toe te kennen.
Informeel betekent dit dat, als A zich in de toestand S bevindt, dat de namen in V aan vooraf gestelde voorwaarden moeten voldoen. Dit proces noemt "S *interpreteren*".

Op een afstand is de *toestand* een redelijk intuïtief gegeven; programmeurs zullen het begrip "toestand" vaak beschrijven als de inhoud van hun programma op een bepaald moment tijdens de uitvoering. Bij het debuggen wordt soms de uitvoering gestopt, en wordt de inhoud van die variabelen dan ook geïnspecteerd.

ASM-toestanden verschillen hiervan, omdat de toestanden allemaal expliciet bepaald worden, en er ook wordt geopereerd vanuit die toestanden zelf, niet van de positie van het programma in de sequentie. Het laat toe om de werking van het algoritme af te richten op basis van de toestanden die optreden.

Het verschil met de λ-calculus (lees: functionele en declaratieve talen) kon niet groter zijn: ASM's hebben een strak beeld van de toestand(en), maar de λ-calculus heeft gewoon **geen** toestand; de S-expressies "bestaan", maar de wereld waarin is vrij van variabelen. De geïnformeerde programmeur zal zelf moeten bepalen welke werkwijze het meest geschikt is voor het op te lossen probleem.

9.1 Initiële toestand

Als de ASM geen module betreft, maar een gewoon programma, dan moet er een bepaalde toestand zijn waarin de ASM begint. In deze toestand worden de waarden gekoppeld aan de identiteiten die voorkomen in de vocabulaire.

De initiële toestand wordt genoteerd als S_0.

9.1.1 De reserve

De toestand van de ASM wijzigt, en soms betekent dat dat er nieuwe elementen aan de functies worden toegekend, of dat er verdwijnen.

Voor externe functies is dit niet moeilijk om te erkennen, maar voor interne functies ligt dit moeilijker. Als we aannemen dat de vocabulaire niet mag groeien, dan zullen we genoegen moeten nemen met wat er al inzit.

Hier is waar het idee van een reserve de intrede doet. De reserve is een element in de vocabulaire waaruit elementen worden gehaald, en anderen in worden opgeborgen.

Niet alle ASM's hebben een reserve nodig. Het A*-algoritme kan bijvoorbeeld volledig opereren zonder nieuwe elementen op te vragen.

Als er toch een reserve is, dan wordt de reserve beschouwd als oneindig [8, proviso, p. 4]. Bedenkt als voorbeeld hiervan een ASM die een graaf voorstelt. Er zijn oneindig veel bogen en knopen die aan een graaf kunnen worden toegevoegd.

Oneindig, en toch beperkt Oneindigheid betekent niet dat *alles* in de reserve zit, integendeel: De inhoud van de reserve is complementair aan de toestand waarin deze zich bevindt.

Uit een abstract oogpunt wordt gezegd dat, als de toestand een nieuw element verkrijgt, dat deze *uit* de reserve gehaald wordt, en vice versa.
Dit impliceert dat, als een functie als argument een element uit de reserve krijgt, deze altijd `false` of `undef` zal teruggeven voor respectievelijk relaties en andere functies.

De onderliggende reden hiervoor is simpel: **De inhoud van de vocabulaire kan niet veranderen**, en dus zijn ook de elementen in de toestand constant. Door te stellen dat de reserve het complement is van de "andere" elementen in de toestand, treedt er geen wijziging op.

9.2 Webserver

We zien al enkele duidelijke toestanden waarin onze webserver zich kan bevinden:

- Er zijn geen verbindingen
- Een *client* vraagt de verbinding aan met de server
- Er zijn één of meerdere verbindingen tot stand gebracht

We kunnen het proces van het bepalen van de huidige toestand in het volgende algoritme gieten:

```
// Handling new and lost connections first
do in-parallel
new-connections
lost-connections
enddo
if ∃x (isConnected(x) = true) then
determine-active-client
elseif activeClient != undef then
communicate
endif
```

Herinner dat er in eenzelfde *run* geen tegenstrijdige updates kunnen gebeuren. Dankzij de invarianten die we eerder hebben opgesteld en het gebruik van een voorwaardelijke regel voor de communicatie, is het duidelijk dat dit niet zal optreden.

In zekere zin is er geen nood om een finale toestand te definiëren. Een webserver is een programma dat in de praktijk niet wordt afgesloten, in tegenstelling tot bepaalde algoritmen.

9.2.1 Reserve van *clients*

Als er een verbinding ontstaat, dan is er dus een bepaalde *client* die daarvoor verantwoordelijk is. Dus definiëren we een reserve `clients` in de webserver.

Hieruit blijkt ook het complementaire aspect van de reserve: De mogelijke *clients* zijn alle computers die een verbinding **kunnen** aangaan met onze webserver, maar die (nog) niet verbonden **zijn**. Vanaf het moment dat ze verbinden, komen ze uit de reserve, en worden ze een "bruikbare" entiteit binnen ons ASM.

Het is ook makkelijk in te zien dat een oneindig aantal elementen niet "alles wat er bestaat" betekent: er zullen misschien wel oneindig veel entiteiten kunnen communiceren met het web, maar een madeliefje zal daar niet tussen zitten. Het is belangrijk om dat te vermelden, omdat zeggen dat iets "oneindig" is nog wel degelijk toelaat om grenzen en aannames te stellen aan onze elementen.

10 Vocabulaire

Beschouw een willekeurig algoritme. Waarschijnlijk zitten daar enkele operaties in, de namen van variabelen, functies die worden opgeroepen, ...
Dat zijn voorbeelden van berekenbare algoritmen. Er zijn ook algoritmen die dat niet zijn, zoals recepten. Ook daar zijn er bepaalde namen die de werking bepalen: roeren, schillen, fruiten, ... Baktijd kan een functie zijn waarvan de parameter de benodigde tijd is.

Dat laatste voorbeeld is natuurlijk niet uit te drukken met een programmeertaal.[8] Maar het vertoont toch gelijkenissen.

Het zijn net die benamingen die volgens de ASM-theorie een integraal onderdeel vormen van het algoritme zelf, en collectief wordt dit de vocabulaire genoemd.

De vocabulaire[9] is een *eindige* verzameling van de functie- en variabelnamen die binnen de ASM bestaan.

De vocabulaire van een ASM kan niet gewijzigd worden. Eens dat de programmeur deze heeft vastgelegd, moet deze zo behouden blijven. Dit houdt verband met het concept van abstractieniveaus, zodat de programmeur verplicht wordt om een niveau te kiezen waarin het probleem correct gemodelleerd kan worden. Blijkt dat de vocabulaire niet toereikend is om het algoritme te beschrijven, dan is het abstractieniveau niet juist, en moet de vocabulaire uitgebreid worden.

[8] Er is natuurlijk technisch gezien niets dat een informaticus tegenhoudt om een programmeertaal te schrijven voor keukenapparaten waarin recepten wel degelijk kunnen worden geschreven als een programma, maar ik heb een sterk vermoeden dat dit (nog?) niet bestaat, en gewone talen worden al eeuwen gebruikt om recepten te schrijven, dus er is ook niet echt een maatschappelijke vraag naar zo'n programmeertaal.

[9] Dit wordt soms ook wel de *signature* van een ASM genoemd.

Waar het bij andere programmeertalen gebruikelijk is dat er doorheen het programma nieuwe variabelen worden aangemaakt (en de kardinaliteit van de toestand van het programma dus kan variëren), is deze in een ASM statisch.[10]

Een vocabulaire wordt genoteerd als Υ.

10.1 Wat als het programma groeit

Het is gebruikelijk voor programma's om te "groeien" naarmate het uitgevoerd wordt. Een compilerprogramma zal bijvoorbeeld constant nieuwe "functies" moeten bijhouden. Vormt de vocabulaire dan geen beperking voor praktische software?

Absoluut niet, integendeel: Deze beperking *helpt* juist om zulke programma's correct te modelleren. Enerzijds wordt de programmeur verplicht om zijn vocabulaire zo te beschrijven dat nieuwe relaties, functies, ... kunnen worden opgevangen binnen de ASM. Anderzijds is het een manier om het juiste AN te kunnen bepalen.

10.2 Inhoud

De vocabulaire beschikt steeds over enkele functienamen:

- `=`
- `true`
- `false`
- `undef`
- $\wedge$
- $\vee$
- $\neg$

Deze functienamen zijn statisch, maar worden "logische namen" genoemd. Met het gebruik van booleaanse operaties moeten de gegeven waarden van booleaans zijn (`true` of `false`), anders moet de operatie `undef` geven.

Omdat ze in elke vocabulaire voorkomen, worden ze niet expliciet vermeld.

10.2.1 `undef`

`undef` heeft speciale toepassingen binnen functies. Het wordt gebruikt om aan te duiden dat er geen kennis is van een bepaalde relatie, maar ook om aan te duiden dat een functie niet weet hoe de invoer te interpreteren.

Dit hangt allemaal af van de context waarin de functie voorkomt. `IsMarried(x,y)` kan `true`, `false` of `undef` zijn, het laatste als er geen zekerheid is.
Maar, stel dat `x = y = 1`, dan kan `undef` ook wijzen op het feit dat de functie

[10] Het kan natuurlijk dat een bepaalde variabele een lijst voorstelt, en de inhoud van die lijst kan aangepast worden in een ASM. Intuïtief lijkt het dan dat de kardinaliteit toch kan groeien, maar voor die variabele blijft een lijst gewoon een lijst, of er nu 0 of 100 elementen in zitten. De kardinaliteit blijft dus constant.

niet weet hoe deze relatie beschouwd kan worden. Getallen kunnen immers niet trouwen. Idem voor `x = y = undef`.

Het kan echter wel zo zijn dat voor bepaalde waarden op voorhand geweten kan zijn of er een relatie mogelijk is. Zo kan `Neighbours(undef, 3)` wel degelijk `false` teruggeven, omdat er geen relatie mogelijk is tussen een `undef` en een getal. Opnieuw, het hangt af van hoe de ASM geschreven wordt, en hoe men redeneert over de gestelde relaties.

11 Uitbreidingen op het ASM-model

Wat tot nu toe beschreven werd, is een model dat werkt voor deterministische, sequentiële algoritmen. Toch zijn er algoritmen die niet zo gemodelleerd zijn.

Enkele mogelijke uitbreidingen van de ASM worden hier toegelicht, alsook de manier waarop dat kan.

11.1 Nondeterminisme

Als we niet een exact vooraf bepaald pad doorheen het algoritme willen doorlopen, maar de keuze willen maken uit een verzameling mogelijkheden, dan spreken we van nondeterminisme. Op het eerste zicht is dit niet direct mogelijk met de formele ASM-definitie, maar in een zekere zin bevat het toch al de vereisten voor deze "uitbreiding":

Bewijs. Stel een ASM A met een externe functie E en een verzameling regels R waarvan we willen dat de uitvoering ervan nondeterministisch is. De waarde van E bepaalt welke regel uit R ook daadwerkelijk uitgevoerd wordt. Nu zou E natuurlijk input kunnen vragen aan de gebruiker, maar voor A maakt het niet uit waar die input vandaan komt. Het is enkel geweten dat E met een bepaalde omgeving kan interageren.
Voor nondeterminisme hoeven we dan gewoon te stellen dat die omgeving niet de gebruiker is, maar iets wat op zichzelf een willekeurig antwoord geeft. Op basis van dat antwoord wordt dan een regel uit R uitgevoerd. □

11.2 Tijd, locatie, ...

Als een algoritme rekening moet houden met de tijd, dan kan dit ook bereikt worden met een externe functie, die telkens de huidige tijd teruggeeft aan het programma. De uitbreiding kan analoog worden toegepast voor een groot aantal variabelen, waarbij de externe functie optreedt als een sensor voor die variabele.

11.3 Webserver

Het spreekt vanzelf dat onze server vaak met meerdere gebruikers tegelijk dient te communiceren. Met deze uitbreidingen kunnen we ook aan die eis tegemoet komen.

We stellen een functie die voor een gegeven gebruiker een tijdstip teruggeeft, namelijk het laatste moment waarop de server met die gebruiker gecommuniceerd heeft. Daarvoor dienen we eerder gemaakte regels ietwat aan te passen.

Naast een externe functie voor de tijd op te vragen (`currentTime()`), introduceren we ook een nieuwe functie `lastUpdate(client)`, die een tijdstip

teruggeeft waarop de laatste update is gebeurd. Nieuwe *clients* krijgen impliciet de waarde `undef` voor hun laatste update.

Het kan natuurlijk voorkomen dat er meerdere *clients* nog niet geüpdatet zijn. Vanaf dat moment kunnen/moeten we gebruik maken van nondeterminisme om toch een keuze te maken uit de mogelijkheden:

```
rule determine-active-client:
let waitingClients = client where lastUpdate(client) is the lowest
   ∧ isConnected(client) = true:
if waitingClients.amount > 1:
activeClient := choose randomly from waitingClients
else:
activeClient := waitingClient
endif
endlet
```

In `communicate` moet er enkel een lijn toegevoegd worden om de tijd op te slaan:

```
rule communicate:
   do in-sequence
   softwareChannel(clientChannel)
   clientChannel(softwareChannel)
   lastUpdate(activeClient) := currentTime()
   activeClient := undef
   enddo
```

12 Voorbeeld: Associatieve arrays

Even vooruit lopen: ASMetaL is een taal die niet ontworpen is voor praktische toepassingen, maar om aan te tonen dat het ASM-model ook in staat is om programma's op een computer te schrijven. Wat betreft theoretische sterkte is het een turingvolledig model.

Opmerkelijk is dat in geen enkel voorbeeldbestand er gebruik gemaakt wordt van een associatieve array, of *map*.

Er is wel een implementatie van het concept van een map, maar deze is zo beperkt dat het gebruik ervan simpelweg niet mogelijk is. De standaardbibliotheek voorziet enkel `merge`, `assign` en `at` als functies die kunnen opereren op het type `Map`.

Nog sterker is dat de enige info over wat mogelijk is voor een Map te vinden is in de Italiaanse versie van de handleiding, waarin vermeld staat dat `Map` niet behoort tot de *construtti supportati*.

Oorspronkelijk werd gedacht dat het ontwikkelen van een ASM voor het A*-algoritme een *priority map* vereist. Later in de thesis wordt een A*-implementatie getoond waar dit toch niet nodig is.

Een *priority map* is net zoals een associatieve *array* een structuur waarbij de elementen (sleutels) gelinkt zijn aan een ander element (waarden). De relatie tussen deze elementen is surjectief.[11]

[11] Een surjectieve relatie van twee verzamelingen (stel A en B) betekent dat elementen uit A slechts aan één ander element van B gerelateerd zijn, maar de elementen uit B kunnen wel aan verschillende elementen uit A gerelateerd zijn.

Het verschilt van een gewone *map* omdat voor deze relaties een bepaalde waarde wordt bijgehouden die als prioriteit dient. Wilt men dus een paar uit de *priority map* halen, dan zal diegene met de hoogste prioriteit gegeven worden.

12.1 ASM

De ASM stelt een PM voor, voor het gemak noemen we het ook zo. De vocabulaire van een PM bevat de volgende functies:

Key Domein van sleutels

Value Domein van waarden

$\mathbb{N}$ De natuurlijke getallen

`dyn` **map(Key)** $\rightarrow$ **Value** Eigenlijke mapping van sleutels op de waarden

`dyn` **priority(Key)** $\rightarrow \mathbb{N}$ Geeft prioriteitswaarde terug van de gegeven sleutel

`rule` **add(Key, Value,** $\mathbb{N}$**)** Voegt een nieuw element toe aan de PM

`rule` **remove(Key)** Verwijdert de sleutel uit de PM

`rule` **update(Key, Value,** $\mathbb{N}$**)** Pas een bestaand element aan in de PM

`stc` **contains(Key)** $\rightarrow \mathbb{B}$ Of de gegeven sleutel al dan niet in de PM zit

`stc` **peek()** $\rightarrow$ **Set(Key)** Geeft de sleutels met de hoogste prioriteit terug

PM heeft geen initiële staat nodig, we kunnen zo ook een formele beschrijving geven.

De regels kunnen als volgt geschreven worden:

```
add(key, value, n) = do
map(key) := value
priority(key) := n
enddo

remove(key) = do
map(key) := undef
priority(key) := undef
enddo

update(key, value, n) = do in-sequence
remove(key)
add(key, value, n)
enddo

contains(key) = if
map(key) != undef then
true
else
false
endif
```

`peek() = (∀x∀y:if priority(x) ≥ priortity(y): x`

`update` is gewoon een combinatie van `add` en `remove`, en vereist dus geen speciale update-regels. De regels `add` en `remove` zijn blokregels waarin de staat van PM parallel wordt aangepast aan de nieuwe elementen.

`contains` is een functie die de staat van PM niet aanpast, maar deze wel inspecteert om een antwoord te formuleren.

`peek` is ietwat gecompliceerder, omdat alle prioriteiten gecontroleerd moeten worden. Voor een formele beschrijving is het echter voldoende om te beschrijven dat we de hoogste waarde willen kennen. Door een *guard* op te stellen die stelt dat we alle prioriteiten willen hebben die groter dan of gelijk zijn aan alle andere prioriteiten, en dat we die sleutels willen kennen.

Wat gebeurt er als `remove` wordt gedaan op een sleutel die niet in PM zit? Merk op dat de update-regels gewoon zullen worden uitgevoerd zonder fouten, de update zal enkel redundant zijn. Idem voor `update`.

Wat als er twee waarden met dezelfde prioriteit zijn? Als de prioriteit de hoogste is, dan worden er gewoon meerdere sleutels teruggegeven in de verzameling. Als het niet de hoogste prioriteit betreft, dan maakt het niets uit voor de output.

Het is belangrijk (een mogelijke invariant!) dat er enkel een prioriteit voor een sleutel kan zijn als die sleutel ook in PM voorkomt. Daarom dat elke regel waar hiermee een aanpassing gebeurt zowel `map` als `priority` aanpast.

13 ASMeta

Van berekingsmodellen worden voor computers programmeertalen ontwikkeld, die programmeurs in staat stellen om het principe ook in de praktijk toe te passen in de vorm van software.

Voor abstract state machines zijn er echter nog niet veel beschikbare talen, en sommige talen worden niet meer onderhouden, waarschijnlijk door een gebrek aan interesse door eindgebruikers.

Een taal die echter wel nog actief onderhouden wordt, is ASMetaL. Deze programmeertaal is ontwikkeld aan de universiteit van Milaan, en geniet nog genoeg aandacht om in deze thesis te bespreken. Deze taal is onderdeel van het ASMeta-framework, dat ontworpen werd om aan te tonen dat ASM's meer zijn dan enkel een informaticatheorie.

13.1 Framework voor een taal

ASMeta is een *framework* dat is opgebouwd uit verscheidene *tools* om het programmeren met ASM's mogeijk te maken. De voornaamste onderdelen worden hier toegelicht.

13.1.1 Syntaxkleuring/AsmEE

Het spreekt voor zich dat er met ASMeta gebruiken een hoop broncode bewerken gemoeid gaat. Daarom dat ASMeta een plugin voor Eclips die syntaxkleuring toevoegt aan Eclipse.

Notepad++ Eclipse is op zich nogal een *bloated* programma, en het zal voor velen vervelend zijn om zo'n programma te installeren om met ASM's te kunnen werken. Voor zijn masterthesis heeft Gianluca Bevilacqua daarom voor Notepad++ een plugin geschreven, zodat programmeurs in het (veel lichtere) Notepad++ ook aan ASM's kunnen schrijven. Spijtig genoeg is het bestand niet meer te vinden op de website.

13.1.2 Animator

De uitvoering van een ASM in het geheugen kan (zeker voor grotere programma's) een probleem vormen tijdens de ontwikkeling. Hierbij kan ASMetaA van pas komen. ASMetaA is enkel beschikbaar als plugin voor Eclipse.

Het idee achter een grafische weergave van de werking van een ASM is dat het toelaat om makkelijker te redeneren over het systeem [2].

Met ASMetaA kan de programmeur elke stap doorheen de ASM manueel uitvoeren. Dat impliceert dus ook dat externe data constant aangeleverd dient te worden, wat ook binnen de animator gebeurt. De vorige toestanden worden ook opgeslagen zodat deze later geraadpleegd kunnen worden.
Er is ook de mogelijkheid om een *random animation* te doen; hierbij zal de animator willekeurig waarden toekennen aan de externe functies. Dit kan handig zijn om het proces te versnellen, omdat men dan niet constant een dialoogvenster krijgt voorgeschoteld om een waarde op te geven.

13.1.3 Visualizer

Naast de animator is er nog een tweede onderdeel dat tracht om op een grafische manier formele modellen ontwikkelen te vergemakkelijken. ASMetaVis laat toe om de onderliggende structuur en relaties binnen de ASM te doorzoeken.

De werking verschilt danig van ASMetaA: De visualizer is niet-interactief en handelt vooral in de structuur van de ASM. De animator laat juist wel interactie toe, en visualiseert het procesverloop van de ASM.

ASMetaVis is eigenlijk een programma dat een gegeven ASM uitleest en de structuur in een boom plaatst. Niet alle elementen worden in hun detail getoond, omdat sommigen juist makkelijker te lezen zijn in tekstformaat dan in een grafiek [1].

Afhankelijk van de complexiteit van de ASM is het soms mogelijk om ook een "semantische visualisatie" te genereren. De paper die ASMetaVis beschrijft, gaat niet verder in op waar de grens ligt voor die complexiteit.

ASMetaVis is enkel als plugin voor Eclipse te gebruiken.

13.1.4 Simulator

De simulator van ASMeta (ASMetaS) is een programma dat toelaat om een gegeven ASM uit te voeren. In principe is dit dus de compiler van ASMetaL, en er zal verder in de thesis ook op die manier naar verwezen worden.

De simulator kan zowel als plugin als *standalone* gebruikt worden.

13.1.5 Verificator

De verificator (ASMetaV) dient om de werking van het ASM te controleren op correctheid. Met andere woorden: ASMetaV dient om de code te testen [4].

De manier waarop dat gebeurt is iets gestructureerder dan voor andere programmeertalen: In plaats van specifiek code te schrijven om functionaliteit te testen, dient de programmeur een zogenaamd "scenario" te geven. De bedoeling hiervan is om te controleren of, gegeven een bepaalde input, ook de juiste output verkregen wordt.
Dit scenario wordt geschreven in AValLa, een taal die speciaal voor dit doel ontwikkeld is. De taal is theoretisch niet zo krachtig als een "echte" programmeertaal, maar dit is expres gedaan, net om onnodige complexiteit te vermijden. In dat opzicht is AvalLa te vergelijken met SQL.

ASMetaV is beschikbaar in twee vormen: Als plugin voor Eclipse, en als een Java-jar.

13.1.6 C++-Transpiler

Hoewel het zich nog in een expermentele fase bevindt[12], is er al gewerkt aan een transpiler voor ASM-code, ASM2C++. De bedoeling van deze *tool* is om ASMetaL-code te vertalen naar C++-code, die dan op Arduino-hardware[13] kan uitgevoerd worden [3].

Het doel hiervan is om via ASM's het mogelijk te maken om programma's beter te modelleren volgens het te beschrijven algoritme, en het in een later stadium te vertalen naar broncode die op een computer kan worden uitgevoerd. Omdat het deel uitmaakt van ASMeta, is het natuurlijk ook mogelijk om de andere *tools* te gebruiken tijdens het ontwikkelingsproces.

Er is echter nog geen *jar* of plugin beschikbaar, maar er zijn al tests uitgevoerd met positieve uitkomsten.

13.1.7 ASMetaL

Centraal binnen het framework is de taal zelf. De taal is duidelijk geschreven met als doel om te experimenteren met de mogelijkheden van ASM's als een programmeertaal. Er zijn dan ook weinig faciliteiten voor het schrijven van software met praktische doeleinden, denk aan *package managers*, gestructureerde documentatie (bv. JavaDoc), maar ook meer informelere zaken zoals programmeerstijl en fora voor de gemeenschap.

ASMeta voorziet in beperkte documentatie over de elementen die binnen de taal bestaan. De volledige syntax van de ASMeta is beschreven volgens een *Extended Backus-Naur Form.*[14] Daarnaast is er ook leesbaardere informatie over de taal, alhoewel niet alles geïmplementeerd is. Er is een kleine standaardbibliotheek die functies aanroept die in Java geschreven zijn.

[12] Op de website van ASMeta wordt Asm2C++ wél beschreven als een "stabiel" onderdeel, maar de code ervoor zit in de *repository* nog altijd onder de experimentele folder.

[13] Technisch gezien is C++-code niet enkel op Arduino-hardware werkzaam, maar ook op gewone computers. Echter, de thesis gaat expliciet over hoe de C++-transpiler gericht is op de sensoren van Arduino's, waarbij bv. I/O via externe functies geïmplementeerd is. De transpiler is dus niet geschikt om voor gewone computers C++-code te genereren.

[14] Alhoewel EBNF voorziet in speciale syntax om bepaalde constructies te noteren, zoals bijvoorbeeld `[]` om een gedeelte aan te geven dat 0 of 1 keer voorkomt, wordt voor ASMeta meer gebruik gemaakt van regex, zoals `()?` om 0 of 1 voorkomen aan te geven.

13.2 Interactie

Voor niet-gesloten algoritmen gebeurt de interactie met de omgeving via externe functies, die *monitored functions* genoemd worden.
Normaal zal ASMeta hiervoor tijdens de simulatie de vraag om input stellen aan de gebruiker, maar het is ook mogelijk om een `.env`-bestand te gebruiken.

Volgens de ASM-theorie moet de waarde van alle externe functies gekend zijn in elke staat. ASMeta zal dan ook elke keer dat de waarde opgevraagd wordt in een regel de gebruiker vragen wat de waarde is.
Dit kan vermeden worden door externe functies zoveel mogelijk te "nesten", zodat de vragen enkel gesteld worden als het ook echt nodig is [11, How to use monitored functions, p. 10].

13.2.1 .env-bestand

De werking van het .env-bestand is niet duidelijk; in de documentatie wordt nergens vermeld hoe en wanneer waarden kunnen worden toegekend.[15]

In de Java-broncode kon ik echter wel een functie vinden die de informatie uit zo'n .env-bestand probeert te halen:

```
public Value readValue(Location location, State state) {
String line;
do {
line = readLine().trim();
} while (line.length()  0 || line.charAt(0)  '#');
Function func = location.getSignature();
try {
return new Parser(line).visit(func.getCodomain());
} catch (InputMismatchException e) {
throw new RuntimeException(e);
    }
}
```

Ik vermoed dat er tijdens elke *run* de waarden één voor één in een bepaalde volgorde aan de externe functies worden doorgegeven. Dit verklaart ook waarom veel voorbeeldbestanden vóór elke waarde een commentaarregel hebben waarin de exacte vraag wordt herhaald. Dit vermoeden wordt bevestigd door de handleiding:

> Usually it's useful to execute a first run of our model in the interactive way, to discover in what order the monitored functions are requested; in fact, it could be difficult to predict the correct order when the code is particularly complex.

13.3 Output

In tegenstelling tot veel andere programmeertalen, heeft ASMeta geen speciale manier om output te printen. Er zijn wel `print`-functies gedeclareerd maar deze

[15] Voor de volledigheid: Het gebruik ervan wordt wel vermeld in de *user guide*, maar echt bruikbare informatie over hoe het werkt is nergens te vinden.

werken niet naar behoren[16]
In plaats daarvan wordt na elke run de inhoud van de staat van het programma afgedrukt. Op die manier kan er natuurlijk wel gecommuniceerd worden naar de gebruiker.

Er is ook geen functionaliteit beschikbaar voor interactie met bestanden.

13.4 Beëindigen van uitvoering

Voor ASM's is er geen vastgestelde manier waarop een programma moet eindigen. Omdat het geen vooraf vastgestelde procedure betreft, moet het programma dus eindigen door interventie van de programmeur.

Voor ASMeta zijn er manieren om dit passief te regelen:

Vooropgesteld aantal runs De programmeur vertelt de compiler dat het programma een aantal state changes moet ondergaan. Hierna stopt het programma automatisch. Voor veel programma's is dit geen oplossing; afhankelijk van de input kan een hoger aantal state changes nodig zijn.

Trigger op invariantie Het kan zo worden ingesteld dat het programma stopt als er een aantal keer achter elkaar geen verandering optreedt in de state. Dit is een stuk handiger, want als de staat niet meer wordt aangepast, dan kan dat betekenen dat het probleem opgelost is, en het programma beëindigd mag worden.

13.5 Andere eigenschappen

ASMeta heeft ingebouwde ondersteuning voor complexe getallen, wat zeer nuttig is voor bepaalde ingenieurstoepassingen en wiskundig onderzoek.

Het is een statische en sterkgetypeerde taal[17] , [18], en leunt op dat vlak dicht aan bij talen zoals Rust en Haskell.

13.6 Beperkingen/Nadelen

De taal bevat enkele punten die bedenkelijk zijn, en toch zeker geen absolute vereiste zijn voor een programmeertaal[19]:

Bestandsnamen Omdat op de eerste lijn in een .asm-bestand altijd de naam van het bestand moet gegeven worden, moet deze naam voldoen aan de definitie van een `<GENERIC_ID>` zoals in de EBNF gegeven [] wat zich beperkt tot het gewone alfabet, cijfers, en het liggend streepje. Symbolen met diakritieken, spaties, leestekens en zelfs gewone streepjes kunnen dus niet in de bestandsnaam gebruikt worden.

[16] In de standaardbibliotheek worden de `print`-functies ook beschreven als "Very nasty functions", wat toch doet vermoeden dat de implementatie ervan nogal ingewikkeld is.

[17] In een statisch getypeerde taal zijn de types die een bepaalde *identifier* kan bevatten gekend tijdens compilatie van de broncode.

[18] Sterk getypeerde talen blokkeren in zekere mate operaties tussen verschillende types, zoals converteringen. In ASMeta kan men bijvoorbeeld geen natuurlijk getal met een geheel getal optellen, omdat dit verschillende types zijn.

[19] Hieronder wordt verstaan: Als een bepaalde eigenschap niet in een andere programmeertaal voorkomt, maar wel in ASMeta, dan doet dat vragen rijzen over de noodzakelijkheid ervan.

Eerste lijn code Deze moet altijd zijn: `(asyncr) asm [bestandsnaam]`. Het is onduidelijk waarom deze vereiste er is, zeker omdat deze informatie al verwerkt zit in de bestandsnaam zelf (uitgezonderd voor `asyncr`, maar dat kan net zo gemakkelijk met een .asma-extensie).

Importeren van standaardbibliotheek Veel talen bieden een basis van functies aan die dan ook direct beschikbaar is voor de programmeur. Het valt op dat bij ASMeta (net zoals bij C) dit toch nog expliciet vermeld moet worden, wat toch een beetje de essentie van "standaard" mist.

13.6.1 Gebrek aan degelijke basis voor algemene software

Ondanks het feit dat er wel een standaardbibliotheek beschikbaar is, zijn de mogelijkheden met ASMeta bijzonder klein, vergeleken met andere talen. Zo is er geen systeem voor bibliotheken uit te wisselen, geen functies om met bestanden te interageren, geen code om met een netwerk te communiceren, ...

Het valt op dat de taal op dit moment vooral dient als een *proof of concept*, om te bewijzen dat *abstract state machines* wel degelijk net zo krachtig zijn als andere berekingsmodellen, en dat er ook functionerende programma's mee geschreven kunnen worden. Dit wordt dan ook bevestigd door de vele *plug-ins* voor Eclipse om een ASM te animeren, te simuleren, ...

13.6.2 Omslachtigheid

Het moet vermeld worden dat ASMeta een relatief omslachtige taal is; het eerste voorbeeld in de *user guide* is een faculteitsberekening.
Het programma telt 35 lijnen code.
Vergelijk met een equivalent programma in Haskell:

```
factorial n
  | n == 0 = 1
  | n > 0  = n * factorial (n-1)
```

Dit is problematisch omdat programmeertalen bestaan voor het gemak van programmeurs; een programma dient volgens de regels van de taal beschreven te worden, en het is van belang dat dit zo gemakkelijk en snel mogelijk kan gebeuren. Een taal moet dus ook streven om het uitdrukken van de werking zo beknopt mogelijk te maken.

Het voorbeeld in Haskell is ook overzichtelijker; vergelijk met de notatie voor wiskundige functies:

$$factorial(n) = \begin{cases} 0 & n = 0 \\ nfactorial(n-1) & n > 0 \end{cases} \quad (1)$$

Hierbij dient vermeld te worden dat algoritmen als voorbeelden slechts een indicatie geven van hoeveel code er uiteindelijk nodig is voor "echte" programma's. Dit kan namelijk opmerkelijk afwijken van de reële complexiteit tijdens het gebruik van de taal.

14 De ASMeta-EBNF

Als programmeertaal beschikt ASMeta over een vooraf gedefinieerde context-vrije grammatica. De structuur waaraan die grammatica moet voldoen wordt voorgesteld middels een *Extended Backus-Naur Form*.

In dit hoofdstuk worden de verschillende sleutelwoorden van deze EBNF toegelicht, en gekoppeld aan de eerder besproken fundamentele ASM-elementen.

Een ASMeta-bestand wordt opgedeeld in drie onderdelen:

ID Waarin de ASM bij naam wordt geïdentificeerd, alsook het type

Header Hier worden de geïmporteerde ASM's geschreven, alsook de vocabulaire en of deze ASM eventueel functies moet exporteren

Body De regels, staten, en de initiële toestand van de ASM

14.1 ID | ASM-identificatie

Over de ID kunnen we kort zijn: Het betreft hier de naam van de ASM, en informatie over het type ASM.

Bij een `asm` worden de gebruikelijke elementen verwacht. Bij `module` wordt er geen initiële toestand verwacht ($S_0 = \emptyset$, en dus ook geen "beginregel".

De aanwezigheid van het *keyword* `asyncr` geeft aan of deze ASM al dan niet asynchroon kan werken.

14.2 Header | Vocabulaire

De header begint met de `ImportClause`.
Dit is vitaal om het principe van abstractieniveaus te kunnen opbouwen; door kleinere ASM's aan te spreken voor wijzigingen die geabstraheerd worden, kan een hoger AN bereikt worden.
Meestal wordt sowieso de standaardbibliotheek geïmporteerd.

Er kan dan ook nog functionaliteit geëxporteerd worden. In de standaardbibliotheek is te zien hoe alle functies beschikbaar worden gesteld voor importering.

14.2.1 `signature`

De vocabulaire van een ASM wordt in het Engels ook wel *signature* genoemd, en zo ook in ASMeta.

Eerst moet de programmeur de domeinnamen bepalen, en daarna de functienamen.
Deze domeinen slaan de brug tussen de abstracte betekenis, en de praktisch toepassingen. Zo blijft het evenwel mogelijk om een `abstract domain` te declareren, maar ook bepaalde getallenreeksen als universum aan te reiken.

In de signature moet voor de verschillende functies aangegeven worden van welk type ze zijn:

`static` Als het een statische functie betreft

`dynamic` Idem voor een dynamische functie

Als er geen type wordt gespecifieerd wordt *default* `dynamic` aangenomen. Indien de functie dynamisch is, moet ook het type gegeven worden:

`monitored` Externe functie

`controlled` Interne functie

`shared` Gedeelde functie

`out` ?

`local` Functies die enkel in de lokale *scope* bestaan

In ASMeta zijn er technisch gezien enkel nullaire en unaire functies, maar door gebruik te maken van `Prod` als type voor het cartesiaans product kunnen *de facto* n-aire functies gedeclareerd worden.

Voor `out` is er geen speciale werking beschreven, en uitvoeren in de simulator geeft ook geen verschil van `controlled`. Vermoedelijk zou een latere versie een dergelijke functie gebruiken om output naar de gebruiker te sturen.

14.2.2 `invariant`

Alhoewel er in de definitie van een ASM geen invariant voorkomt, kent ASMeta toch een manier om deze te bepalen. Voor elke invariant dient de programmeur over een bepaalde naam een term op te geven waaraan deze moet voldoen. Het programma stopt direct als de invariant niet meer geldig is.

14.2.3 `derived`

Een "afgeleide" functie is een speciale vorm van een statische functie, waarvan de output volledig afhangt van de parameters. Een gewone statische functie kan variëren op basis van de toestand, maar voor eenzelfde staat wordt wel dezelfde output gegeven.

14.2.4 Infix of prefix?

Eerder in de thesis werd opgemerkt dat er niet vastgelegd is welke functienamen als pre- of infix geschreven dienen te worden. ASMeta doet dit wel, en beperkt de infixnotatie tot de bekende wiskundige bewerkingen, vergelijkingen en logische operatoren.[20]

14.3 Body | Definities en regels

Na de declaratie volgt de definitie van de vocabulaire.

De programmeur dient eerst de domeinen te definiëren en af te bakenen. Als dat niet gebeurt voor een domein uit de vocabulaire, wordt het volledige domein aangenomen. Zeker voor types waarvan het domein niet van belang is, is dit een handige optie.

Hierop volgt de definitie voor **statische** functies. Dynamische functies worden pas gedefinieerd in de initiële toestand, dus die worden hier achterwege gelaten.

[20] De volledige lijst kan worden geraadpleegd op `http://fmse.di.unimi.it/asmeta/download/AsmetaL_quickguide.html#infixOp`.

De declaratie van regels vindt hier ook plaats, omdat regels niet tot de vocabulaire behoren. Ze worden dus direct gevolgd door hun definitie.
Er zijn twee soorten regels in ASMeta: `macro`- en `turbo`-regels. Het verschil ligt in de structuur van de regels, een turboregel zijn gecomposeerde regels, zoals de blokregel.

Op het einde is het nog mogelijk om invarianten aan te wijzen voor termen. Deze blokkeren de uitvoer van het programma als er na een *run* niet meer aan voldaan wordt.

Hierna volgen ook nog constructies, beschreven als `JusticeConstraint` en `FairnessConstraint`, maar het bestaan hiervan is nergens gedocumenteerd in andere bronnen, laat staan hoe ze werken of waarvoor ze dienen.

14.4 Programmatuur | Toestanden

Na de *body* volgen nog twee constructies die niet vereist zijn als het een module betreft: `main` en `default init s0`.

`main` is de regel die aan het begin van elke *run* start. Het is de bedoeling dat vanuit deze regel de andere regels bereikt kunnen worden.

In zekere zin gebeurt het definiëren van de **toestanden** van de ASM hier; door de voorwaarden voor elke regel uit te schrijven, kan worden bepaald welke regel vanuit welke toestand(en) wordt uitgevoerd.

Na de definitie van de staten, wordt de initiële staat geïnitialiseerd.

14.5 Addendum: Italiaanse documentatie

De documentatie van ASMeta is nogal versnipperd. Voor sommige elementen was de informatie zo schaars dat deze enkel in het Italiaans te vinden was.

Uit het Italiaanse document [12] kon worden opgemaakt dat een hoop constructies eigenlijk **niet** werken in ASMeta. Deze zijn:

- Complexe getallen
- `char`
- `Rule` als domein
- `Bag`
- `BagTerm`
- `BagCt`
- `Map`
- `MapTerm`
- `MapCt`
- `ExistUniqueTerm`

14.6 Conclusie

De fundamentele elementen van de ASM zijn allemaal vertegenwoordigd in de grammatica van ASMeta.

Opvallend is de rare terminologie die gebruikt wordt, zoals `macro` en `turbo` voor regels, `out` en `local` voor zaken die niet in de gewone ASM-definitie voorkomen. Vermoedelijk zijn sommige constructies voorzien voor de compiler.

Er zijn echter ook constructies die simpelweg nog niet geïmplementeerd zijn, en waar ook geen informatie over te vinden valt, behalve dat ze niet werken. De reden hiervoor is niet duidelijk, een reden wordt nergens gegeven.

15 Voorbeeld: Het A*-algoritme

Ter demonstratie van de verschillen die ASM's hebben ten opzichte van het Turingmodel en de lambdacalculus, wordt in dit hoofdstuk een bekend algoritme geschreven in de drie modellen.

Het A*-algoritme is een algoritme dat in een graaf het kortste pad zoekt, maar toelaat aan de gebruiker ervan om een heuristiek te gebruiken die het zoeken sneller laat plaatsvinden. In sé betekent dit dat het algoritme van Dijkstra dus gewoon een versie van A* is met een lege heuristiek.

Dit algoritme is complex genoeg om de verschillen tussen de verschillende modellen te illustreren, maar toch nog begrijpelijk genoeg om niet te struikelen over moeilijke concepten.

Input Een ongerichte, gewogen graaf; beginknoop; doelknoop; heuristische functie.

Output Het kortste pad tussen de begin- en doelknoop in de graaf.

Turingmodel Python

λ-calculus Clojure

ASM ASMeta

Er is voor Python en Clojure gekozen omdat deze talen veel procedures abstraheren die niet inherent zijn aan het A*-algoritme, zodat een gelijkaardig abstractieniveau kan worden gesimuleerd. Voor ASMeta is gekozen omdat het de enige praktische optie is.

15.1 Turingmodel - Python

Het voorbeeld voor het Turingmodel is een aangepaste versie van de versie van RedBlobGames.

```
from queue import PriorityQueue
import graph

def Astar(sourceNode, target_node, graph, heuristic):
    frontier = PriorityQueue()
    frontier.put(sourceNode, 0)
    came_from = dict()
```

```
cost_so_far = dict()
came_from[sourceNode] = None
cost_so_far[sourceNode] = 0

while not frontier.empty():
current_node = frontier.get()

if current_node == target_node:
    break

for neighbor_node in graph.neighbors(current_node):
    new_cost = cost_so_far[current_node]
    + graph.cost(current_node, neighbor_node)
    if neighbor_node not in cost_so_far
       or new_cost < cost_so_far[neighbor_node]:
        cost_so_far[neighbor_node] = new_cost
        priority = new_cost
                 + heuristic(target_node, neighbor_node)
        frontier.put(neighbor_node, priority)
        came_from[neighbor_node] = current_node
```

Onmiddelijk is duidelijk dat we voor Python een module moeten inladen omwille van het AN dat we wensen te benaderen.

Omdat in het Turingmodel vooral een sequentiële aanpak wordt afgedwongen, is het hele algoritme ook als zodoende geschreven. Het imperatieve paradigma contrasteert ook sterk met Clojure; doorheen het algoritme worden aanpassingen aan de variabelen doorgevoerd.

15.2 λ-calculus - Clojure

Dit voorbeeld komt uit een zelfgeschreven bibliotheek voor grafen.

```
(defn A*
  ([graph
    heuristic
    source-node
    target-node
    frontier-nodes
    previous-nodes
    path-distances]
   (if (or (empty? frontier-nodes) (contains? previous-nodes
                                               target-node))
     {:frontier frontier-nodes
      :previous previous-nodes
      :distance path-distances}
     (let [neighbor-nodes
           (filter #(nil? (get path-distances %))
                   (neighbors graph source-node))
           current-node source-node
           neighbor-costs
           (map #(+
```

```
                (get path-distances source-node)
                (get (:weights graph) [source-node %])
                (heuristic % target-node)) neighbor-nodes)
        neighbors-mapping
        (zipmap neighbor-costs neighbor-nodes)]
    (let [next-node
          (get neighbors-mapping
               (reduce min (keys neighbors-mapping)))
          new-frontier (put-prior
                        (remove-prior frontier-nodes)
                        next-node
                        (reduce min
                          (keys neighbors-mapping)))]
        (recur graph
               heuristic
               (prior new-frontier)
               target-node
               new-frontier
               (assoc previous-nodes next-node source-node)
               (assoc path-distances next-node
                  (reduce min (keys neighbors-mapping)))
               ))))))
```

Clojure heeft veel structuren uit de discrete wiskunde geïmplementeerd in de taal zelf. Daarom is het A*-algoritme het makkelijkst te beschrijven door ook in te spelen op die abstracte structuren, en ze ten volle te benutten.

Omdat het een dynamische taal is (en de types dus niet op voorhand gekend zijn), maakt het ook niet veel aannames over de gewichten van de bogen; zolang ze maar vergelijkbaar en optelbaar zijn.

De constant wijzigende staat van het A*-algoritme verplicht het algoritme in Clojure wel om voor praktisch elke wijziging in de staat de functie recursief opnieuw aan te roepen. Dit, omdat Clojure hamert op *immutabiliteit*; een functie uitvoeren geeft steeds een nieuwe waarde terug.

Het ombouwen van A* naar een recursieve versie toont wel aan dat dit de leesbaarheid negatief kan beïnvloeden.

15.3 ASM - ASMeta

15.3.1 Code

In ASMeta moet de gebruiker informatie verschaffen over de graaf. Daarom dat de functies die informatie opvragen uit de graaf allemaal extern gedeclareerd zijn.

Het is in ASMeta echter niet interessant om dezelfde structuren te gebruiken als Clojure/Python; de taal kent geen bruikbare `Map`, en de generalisatie van functies laat een werkwijze toe die dichter tegen de essentie van het algoritme aanleunt.

```
asm astar

import ASMeta/StandardLibrary
```

```
signature:

domain Node subsetof Integer

// Graph related functions
// Returns weight of edge
dynamic monitored distance: Prod(Node, Node) -> Integer
// between given nodes. Returns undef if there's no edge between them.
// Info about the neighbors comes from graph => extern
dynamic monitored neighbors: Node -> Powerset(Node)

// Externally provided heuristic
monitored heuristic: Prod(Node, Node) -> Integer

// The priority holder for the nodes
dynamic controlled priority: Node -> Integer

// Returns the previously visited node, or undef if N/A
dynamic controlled previous: Node -> Node

// RELATION - Returns whether Node was already visited
dynamic controlled visited: Node -> Boolean
dynamic controlled costSoFar: Node -> Integer

dynamic monitored frontier: Node

// Variables
dynamic controlled currentNode: Node
dynamic controlled sourceNode: Node
dynamic controlled targetNode: Node
dynamic controlled first_run: Boolean



definitions:


macro rule r_finish = skip

turbo rule r_newCurrentNode =
let ($newNode = frontier) in seq
previous($newNode) := currentNode
currentNode := $newNode
// What follows doesn't necessarily have to happen sequential
visited(currentNode) := true // Block future visitations
priority(currentNode) := undef // Remove from priority queue
endseq
endlet
```

```
turbo rule r_exploreCurrentNode =
forall $neighbor in neighbors(currentNode) with true do
let ($new_cost = costSoFar(currentNode) + distance(currentNode, $neighbor)) in
if (not (visited($neighbor) = true)) then
par
costSoFar($neighbor) := $new_cost
priority($neighbor) := $new_cost + heuristic($neighbor, targetNode)
previous($neighbor) := currentNode
endpar
else if (isDef(costSoFar($neighbor))) then
if ($new_cost < costSoFar($neighbor)) then
par
costSoFar($neighbor) := $new_cost
priority($neighbor) := $new_cost + heuristic($neighbor, targetNode)
previous($neighbor) := currentNode
endpar
endif
endif
endif
endlet

macro rule r_setPriority = par
priority(sourceNode) := 0
costSoFar(sourceNode) := 0
previous(sourceNode) := sourceNode
first_run := false
endpar


main rule r_main = seq
if (first_run = true) then
r_setPriority[]
endif
r_newCurrentNode()
if (currentNode = targetNode or isUndef(currentNode)) then
// currentNode will be undef if the frontier is empty,
// meaning the search can't continue
r_finish[]
else
r_exploreCurrentNode()
endif
endseq

default init s0:
// Need to define currentNode so the self-pointer relation
// in r_newCurrentNode will occur
function sourceNode  = 1//<SOURCE NODE>
function targetNode  = 9//<TARGET NODE>
function currentNode = 1//<SOURCE NODE>
function first_run = true // Necessary for priority node setting
```

Omdat ASMeta een statische taal is (in tegenstelling tot Clojure en Python), moet het type van de namen in de vocabulaire op voorhand gekend zijn. Voor de knopen werd origineel `Any` gekozen, zodat de ASM gebruikt kan worden voor eender welk soort structuur van knopen. Achteraf is gekozen om de knopen als `Integers` te beschouwen om niet al te veel problemen te veroorzaken met het framework zelf.

Functies met een niet-unaire ariteit worden *technisch gezien* niet ondersteund in ASMeta, maar de taal lost dat op door te stellen dat de gegeven parameter een cartesiaans product vormt over de gegeven termen. Qua schrijfwijze in het algoritme merkt men daar weinig van, maar dit is de reden waarom in de declaratie `Prod(X, Y)` gebruikt wordt.

De afstand wordt bepaald via een externe functie, `distance(NodeA, NodeB)`. Omdat A* een algoritme is dat het kortste pad berekent tussen knopen in een graaf, is gesteld dat `distance` `undef` teruggeeft als de gegeven knopen geen buren zijn van elkaar (lees: er moet een boog zijn tussen de knopen om de afstand te kennen).

Hoe wordt uiteindelijk uit deze ASM het pad bepaald? Men hoeft enkel de functie `previous(Node)` uit te lezen vanaf `currentNode`, en door te itereren met de knoop die wordt gegeven. De knoop voorafgaand aan de beginknoop werd in S_0 gelijkgesteld aan zichzelf, zodat, als dezelfde knoop wordt teruggegeven, het volledige pad beschreven is.[21]
Dit betekent ook dat als in de laatste staat `currentNode` niet gelijk is aan `targetNode`, er geen pad bestaat tussen de begin- en eindknoop.

15.3.2 Formele beschrijving

Vocabulaire en invarianten/aannames Deze aannames blijven onveranderd gedurende de uitvoer van de ASM:

- Het heuristisch algoritme is "admissible".
- De graaf is statisch tijdens de uitvoering van het programma.
- De externe functies geven steeds de correcte informatie terug.

De vocabulaire bevat de volgende namen:[22]

Node Abstract domein dat een knoop in de graaf voorstelt.

`ext` **frontier → Node** Geeft de knoop terug uit de frontier met de hoogste prioriteit.

[21] Normaal gezien zou `undef` gebruikt worden om het einde van het pad aan te geven, maar in deze functie wordt die waarde al gebruikt om de logische betekenis aan te geven: dat er geen knoop voor de gegeven knoop ontdekt werd.

[22]

`dyn` Dynamisch

`stc` Statisch

`rel` Relatie

`ext` Extern

`rule` Regel

`stc` **sourceNode → Node** De bronknoop; waar het pad begint.

`stc` **targetNode → Node** De knoop waarnaar het kortste pad gevonden dient te worden.

`dyn` **currentNode → Node** De knoop die op dit moment onderzocht wordt in het algoritme.

`dyn rel` **Visited(Node) → $\mathbb{B}$** Of `Node` al bezocht werd of niet.

`stc ext` **Distance(Node, Node) → $\mathbb{N}$** Geeft het gewicht van de boog tussen de gegeven knopen.

`stc ext` **Neighbors(Node) → Set(Node)** De verzameling van alle knopen die de buur zijn van `Node`.

`dyn rel` **priority(Node) → $\mathbb{N}$** De prioriteit van de Node.

`dyn` **previous(Node) → Node** Geeft de knoop terug die voorafging aan `Node` in het onderzoek.

`dyn` **costSoFar(Node) → $\mathbb{N}$** De som van de gewichten geteld van de bronknoop tot `Node`.

`ext` **heuristic(Node, targetNode) → $\mathbb{N}$** De heuristieke functie.

`rule` **finish** Uitgevoerd als het algoritme klaar is.

`rule` **newCurrentNode** Wijzigt de staat van de ASM zodat de volgende knoop voor onderzoek ingesteld wordt.

`rule` **exploreCurrentNode** Regel waarin het onderzoek plaatsvindt.

`rule` **main** Uit te voeren constructie bij het begin van iedere *run*.

`rule` **setPriority** Extensie van instellen van S_0.

S_0 S_0 is de grondstaat van de ASM, en beschrijft de staat waarin het programma begint:
S_0 stelt de bron-, doel- en huidige knoop in, respectievelijk `sourceNode`, `targetNode` en `currentNode`. Het algoritme start vanaf de bronknoop, dus in S_0 : $sourceNode = currentNode$.

```
ground state S0:
do
sourceNode := <SOURCE NODE>
targetNode := <TARGET NODE>
currentNode := <SOURCE NODE>
priority(sourceNode) := 0
costSoFar(sourceNode) := 0
previous(sourceNode) := <SOURCE NODE>
enddo
```

Regels

main Zoals eerder vermeld wordt deze regel uitgevoerd bij de start van de *run*. Er wordt gecontroleerd of de voorwaarden voldaan zijn voor het beëindigen van het algoritme, anders wordt `exploreCurrentNode` uitgevoerd.

newCurrentNode Deze regel wijzigt de staat van de ASM zodat de volgende knoop kan worden onderzocht.
Vooraleer `currentNode` wordt aangepast, wordt `previousNode` zo ingesteld dat de knoop van de vorige *run* wordt aangeduid als de vorige knoop.
Daarna wordt de staat zo aangepast dat de volgende knoop uit de `frontier` onderzocht wordt.

exploreCurrentNode Op deze regel wordt gekeken naar de buren van `currentNode`. Als het een onbezochte knoop betreft, dan worden de heuristiek toegepast, en daarna mee in de frontier geplaatst, zodat deze kandidaat is voor de volgende run. Knopen die niet het onderzoeken waard zijn, worden overgeslagen.

finish Dit is een `skip`-regel. Als deze wordt uitgevoerd in ASMeta, wordt de simulator ingelicht dat de *update set* van deze run leeg is, wat het programma beëindigt.

$$finish \ = \phi : \phi S = S$$

setPriority Het instellen van de initiële toestand laat niet toe om op voorhand waarden toe te kennen aan functies die een parameter vereisen. Deze regel wordt eenmalig in het begin opgeroepen om rond deze beperking heen te werken.

15.3.3 Debugging

Het programma beschrijven vereistte ook debugging, hetgeen op te delen valt in **compileren** en **runtime**.

Als er fouten optreden tijdens het compileren, dan geeft de compiler duidelijk aan op welke lijn welke fout werd ontdekt. Dit ging dan ook snel genoeg.

Een heel ander verhaal is het debuggen tijdens runtime-fouten, waarin ASMeta enorm tekortschiet:

Het volledige ASMeta-framework is geïmplementeerd in Java, en dat betekent dat ASM's ook in die implementatie worden uitgevoerd. Als er echter een fout optreedt in de uitvoering, dan wordt er een *stack trace* uitgeprint van *de Java stack*, en niet van het ASM in kwestie. Dit komt er uiteindelijk op neer dat niet-syntaxgerelateerde fouten zonder enige hulp door de programmeur volledig zelf dienen te worden gevonden én opgelost.

Dit, samen met de beperkingen in beschikbare documentatie, maken een goed werkend programma schrijven een zeer zware opdracht, zeker omdat het niet zo zwaar moet zijn. De compiler voorziet wel in een *debug*-optie, maar die geeft enkel informatie om fouten in de compiler *zelf* te vinden.

Een voorbeeld van een fout die optreedt tijdens *runtime* (61 lijnen overgeslagen):

```
java.lang.IllegalArgumentException
at org.asmeta.simulator.value.UndefValue.equals(UndefValue.java:43)
```

```
at java.util.Arrays.equals(Arrays.java:2829)
at org.asmeta.simulator.Location.equals(Location.java:111)
...
at org.asmeta.simulator.TermEvaluator.visit(TermEvaluator.java:289)
...
at org.asmeta.simulator.TermEvaluator.visit(TermEvaluator.java:260)
...
at org.asmeta.simulator.RuleEvaluator.visit(RuleEvaluator.java:240)
...
at org.asmeta.parser.util.ReflectiveVisitor.visit(ReflectiveVisitor.java:56)
...
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
...
at org.asmeta.simulator.main.AsmetaS.runWith(AsmetaS.java:88)
...
at org.eclipse.jdt.internal.jarinjarloader.JarRsrcLoader.main
```

De output doet *vermoeden* dat er ergens een parameter van een verkeerd type wordt gebruikt, alhoewel dat hoogstens een geïnformeerde gok is.

15.3.4 Test: graaf 1

Graaf 1 is een simpele graaf van een reeks numerieke knopen:

Door gebruik te maken van de simulator is de werking van het algoritme getest. De volledige output kunt u vinden in appendix A.

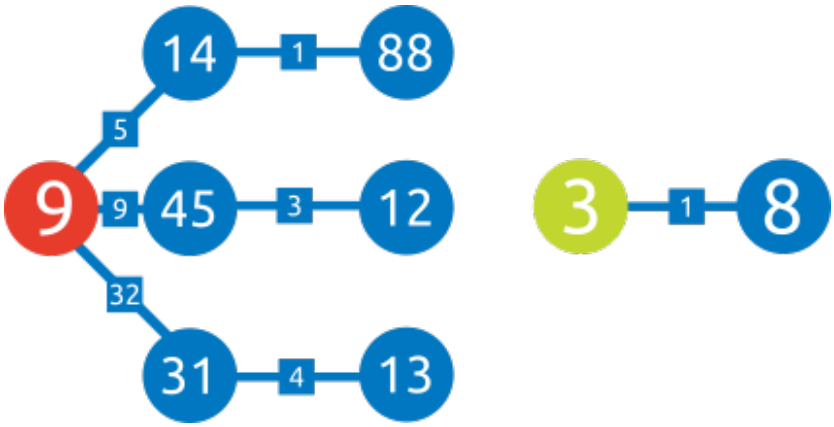
De toestand van het programma wordt (voor deze simpele graaf) op de juiste manier aangepast tijdens elke run. Vooral belangrijk is dat de prioriteit correct wordt aangepast, en dat bezochte knopen ook `undef` als prioriteit krijgen toegewezen, zodat ze niet direct terug in de frontier belanden.

De laatste *runs* zijn ook belangrijk: Nadat de doelknoop bereikt is, stopt het algoritme met verder te vragen, wat erop wijst dat het (zoals bedoeld) erkent dat het in de toestand is waarin de knoop gevonden is.

`previous` wordt ook naar behoren bijgehouden. Op het einde van de uitvoering is dan ook een pad te trekken door omgekeerd (dus vanaf de doelknoop) terug te gaan tot men bij de bronknoop aankomt.

15.3.5 Test: graaf 2

Graaf 2 is ietwat anders:



Er is geen pad tussen de bron- en doelknoop, wat impliceert dat A* alle bereikbare knopen onderzoekt in een poging tot bij het doel te geraken. Dat gebeurt ook, minus de knopen die niet verbonden zijn met bogen naar de bronknoop.

Deze graaf is ook ongericht, alhoewel terugkeren geen optie is, omdat bezochte knopen in `visited` worden aangemerkt. Dat is ook de bedoeling; een pad kan nooit korter worden door eenzelfde boog meermaals te volgen[23].

Om te voorkomen dat het algoritme een bepaalde volgorde van knopen zou prefereren, zijn de getallen willekeurig gekozen.

15.3.6 Grotere grafen

Om het algoritme goed te kunnen testen, zou het mogelijk moeten zijn om zeer grote grafen aan te geven aan het programma. Dit gaat jammer genoeg niet al te vlot: Doordat de graaf extern wordt aangeleverd, moet er bij elke run een hoop informatie manueel door de programmeur worden aangeleverd. Dat is praktisch niet te doen.

.env? Eerder in de thesis is kort het .env-bestand besproken, waarin het mogelijk is om een bepaald scenario te doorlopen door automatisch input te voorzien als het programma daarom vraagt.

Dit zou normaal moeten toelaten om grotere grafen te automatiseren door deze via zo'n bestand in te voeren. Helaas maakt de ASM gebruik van verzamelingen, en hierin is geen volgorde van welke elementen eerst worden opgevraagd. Dat betekent dat de .env-bestanden niet gebruikt kunnen worden, omdat dit kennis vereist van de volgorde waarin informatie opvraagd wordt. Een andere manier voor input (zoals een eigen bestand inlezen) is ook niet mogelijk, omdat ASMeta hiervoor geen faciliteiten heeft.

15.4 Conclusie

Het valt op dat met het ASM minder informatie nodig is over de graaf zelf, en dat de functies op zich al krachtig genoeg zijn, zodat structuren van een hogere orde niet nodig zijn.

[23] Aangenomen dat er enkel positieve gewichten in de boog bestaan, maar dat is sowieso een vereiste om A* te kunnen toepassen binnen een graaf.

Daar staat wel tegenover dat het praktisch nut van de andere twee talen hoger is, omdat die programmeertalen niet gebouwd zijn om een wetenschappelijke theorie aan te tonen.

De manier hoe de grafen geïnspecteerd worden verschillen ook danig: Python behandelt de graaf als een speciale structuur waaruit specifieke functies kunnen worden opgevraagd.
Clojure ziet een graaf als een soort *mapping* en infereert alle benodigde informatie *on-site* van die ene graaf.
Met ASMeta is er technisch gezien geen graaf nodig. Omdat de graaf van buitenaf wordt geleverd, worden enkel de benodigde (externe) functies in de vocabulaire opgenomen. Dit benadert de gewenste abstractie beter; het is niet nodig om met een `PriorityQueue` te werken, of een aparte graafstructuur te beschrijven.

Echter: Van de drie versies is de ASM-versie misschien wel de minst intuïtieve. De structuur van de code verschilt enorm fel met de werking ervan. Het is ook het langste van de drie.
Men kan ook argumenteren dat de graaf benaderen door externe functies niet erg pragmatisch is, maar voor de werking van het algoritme zelf maakt dat weinig verschil. De eerder aangehaalde omslachtigheid komt ook naar boven; op het eerste zicht simpele opdrachten vereisen omwegen in de code om uitgevoerd te krijgen.

16 Conclusie

Waar liggen de grenzen van de informatica? Met de ASM-thesis is het duidelijk dat het beschrijven van algoritmen zeker niet beperkt blijft tot binaire getallen. Zolang dat er informatie aanwezig is, kunnen (en zullen) informatici spelen met de mogelijkheden. Met abstract state machines hebben ze een extra speeltje gekregen.

Het begrip van abstractieniveau heeft mij persoonlijk het meest verbaasd. Hoe één rekenmodel in staat is om de abstractie per algoritme aan te passen is enorm opzienbarend.

Ik blijf na deze bachelorproef toch met een kleine leegte achter: Ik zou graag deze technieken gebruiken in een programma, maar op dit moment is de enige mogelijkheid ASMeta, en dat is spijtig genoeg ontoereikend voor echt hobbywerk. Ik vrees dat dit tekort aan praktische implementaties de populariteit van ASM's tegenwerkt. Hopelijk verandert dit in de nabije toekomst. Zeker omdat het nu al gebruikt wordt voor bepaalde formele specificaties, ben ik hoopvol dat ASM's ook een doorbraak kunnen maken naar meer praktische velden, en niet enkel een theorie blijven die zelfs onder informatici relatief onbekend zijn.

Ik zie in ASM's een behulpzame techniek voor industriële toepassingen, waarin grote processen en complexe software veel moeite en onderhoud vereisen. Een model dat toelaat om specificatie en berekening met elkaar wiskundig te combineren zou hier een aanwinst van onschatbare waarde kunnen zijn.

Persoonlijk denk ik dat een praktische doorbraak van ASM's mogelijk wordt als er een programmeertaal ontwikkeld wordt die inhaakt op een bestaand ontwikkelingsplatform, zoals de JVM. Het kost namelijk te veel tijd en moeite om al die functionaliteit (websockets, I/O, geheugenbeheer, ...) die al in andere talen bestaat, speciaal voor een nieuwe taal opnieuw te schrijven. Dit proces is

al eens toegepast voor Clojure, en dat liet toe om al bestaande programma's die in Java geschreven waren, gradueel om te vormen tot functionele software. Als dat mogelijk zou zijn om met ASM's te doen, dan zal er langzaam maar zeker een ingang worden gevonden voor ASM's in het bedrijfsleven, en van daaruit ook een terugkeer naar de schoolbanken, als blijkt dat bedrijven nood hebben aan informatici die in deze materie geschoold zijn.

17 Dankwoord

Dank aan Yuri Gurevich voor zijn bedenkingen waarop dit hele proefwerk is gebaseerd. Zijn werk heeft een grote positieve bijdrage geleverd aan de wetenschap.

Vanzelfsprekend gaat ongelofelijk veel dank uit aan mijn promotor, prof. dr. Jan Van den Bussche, die mij vanaf het begin enthousiast heeft willen begeleiden in het ontdekken van deze materie. Ik heb veel gehad aan zijn hulp, die hij me proactief heeft aangereikt.

Ik wil ook deze kans benutten om de naasten in mijn leven te bedanken voor de onvoorwaardelijke steun die ik nodig heb gehad om aan deze thesis te kunnen werken, en daarmee in het bijzonder mijn vriend Jonathan.

Referenties

[1] Paolo Arcaini e.a. „Visual Notation and Patterns for Abstract State Machines". In: *Software Technologies: Applications and Foundations*. Red. door Paolo Milazzo, Dániel Varró en Manuel Wimmer. Cham: Springer International Publishing, 2016, p. 163–178. ISBN: 978-3-319-50230-4. URL: https://cs.unibg.it/gargantini/research/papers/HOFM2016_asmPatterns.pdf.

[2] Silvia Bonfanti, Angelo Gargantini en Atif Mashkoor. „AsmetaA: Animator for Abstract State Machines". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Red. door Michael Butler e.a. Cham: Springer International Publishing, 2018, p. 369–373. ISBN: 978-3-319-91271-4. URL: https://cs.unibg.it/gargantini/research/papers/abz2018_Animator.pdf.

[3] Silvia Bonfanti e.a. „Asm2C++: A Tool for Code Generation from Abstract State Machines to Arduino". In: *NASA Formal Methods*. Red. door Clark Barrett, Misty Davies en Temesghen Kahsai. Cham: Springer International Publishing, 2017, p. 295–301. ISBN: 978-3-319-57288-8. URL: https://cs.unibg.it/gargantini/research/papers/asm2cpp_nasafm17.pdf.

[4] Alessandro Carioni e.a. „A Scenario-Based Validation Language for ASMs". In: *Abstract State Machines, B and Z*. Red. door Egon Börger e.a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 71–84. ISBN: 978-3-540-87603-8. URL: https://cs.unibg.it/gargantini/research/papers/abz08_avalla.pdf.

[5] Jos De Greef. „Luchtvaartverkeer op Schiphol nabij Amsterdam gehinderd door technische storing". In: *VRT NWS* (15 aug 2018). URL: https://www.vrt.be/vrtnws/nl/2018/08/15/luchtvaartverkeer-op-schiphol-nabij-amsterdam-ligt-stil-door-tec/ (bezocht op 08-01-2019).

[6] Stefan Grommen. „Vijf dagen technische problemen bij Argenta: hoe kan zoiets?" In: *VRT NWS* (6 apr 2018). URL: https://www.vrt.be/vrtnws/nl/2018/04/06/vijf-dagen-technische-problemen-bij-argenta/ (bezocht op 08-01-2019).

[7] Yuri Gurevich. „Evolving Algebras 1993: Lipari Guide". 2018. arXiv: 1808.06255. URL: http://web.eecs.umich.edu/~gurevich/Opera/103.pdf (bezocht op 08-01-2019).

[8] Yuri Gurevich. „May 1997 Draft of the ASM Guide". In: mei 1997. URL: http://web.eecs.umich.edu/~gurevich/Opera/129.pdf (bezocht op 08-01-2019).

[9] Yuri Gurevich. „The Sequential ASM Thesis". In: deel 67. Feb 1999, p. 93–124. URL: http://web.eecs.umich.edu/~gurevich/Opera/136.pdf (bezocht op 08-01-2019).

[10] Trui De Maré. „Problemen bij apotheken met elektronische voorschriften". In: *VRT NWS* (12 jun 2018). URL: https://www.vrt.be/vrtnws/nl/2018/06/12/problemen-bij-apotheken-met-elektronische-voorschriften/ (bezocht op 08-01-2019).

[11] FMSE LAB - University of Milan. *ASMetaL - A user guide*. URL: http://hfmse.di.unimi.it/asmeta/download/AsmetaL_guide.pdf (bezocht op 08-01-2019).

[12] FMSE LAB - University of Milan. *Piccolo manuale d'utilizzo del simulatore AsmetaS*. Italian. URL: http://fmse.di.unimi.it/asmeta/download/AsmetaS_quickguide_it.pdf (bezocht op 08-01-2019).

[13] Bart Rooms. „Telenetkijkers tijdlang zonder digitale televisie". In: *VRT NWS* (11 jun 2018). URL: https://www.vrt.be/vrtnws/nl/2018/06/11/telenetkijkers-tijdlang-zonder-digitale-televisie/ (bezocht op 08-01-2019).

[14] Kristiaan Grauwels Tomas Teetaert Katrien Kubben. „Uw job overgenomen door artificiële intelligentie?" In: (19 sep 2018). URL: https://www.vrt.be/vrtnu/a-z/pano/2018/pano-s2018a12/ (bezocht op 08-01-2019).

[15] Michaël Torfs. „Treinverkeer in Amsterdam en omstreken grondig verstoord". In: *VRT NWS* (21 aug 2018). URL: https://www.vrt.be/vrtnws/nl/2018/08/21/treinverkeer-in-amsterdam-en-omstreken-grondig-verstoord/ (bezocht op 08-01-2019).

[16] Ludwig De Wolf. „Geen WK kijken in enkele straten in Oelegem door mysterieuze storingen". In: *VRT NWS* (4 jul 2018). URL: https://www.vrt.be/vrtnws/nl/2018/07/04/geen-wk-kijken-in-enkele-straten-in-oelegem-door-mysterieuze-sto/ (bezocht op 08-01-2019).

18 Appendix A: Graaf 1

```
file successfully parsed for asm astar
Insert a constant in Node of type Integer for frontier:
1
Insert a set {} of constant in Node of type Integer for neighbors(1):
Insert a set:
{2}
Insert a integer constant for distance(1,2):
1
Insert a integer constant for heuristic(2,9):
0
<State 1 (controlled)>
costSoFar(1)=0
costSoFar(2)=1
currentNode=1
first_run=false
previous(1)=1
previous(2)=1
priority(1)=undef
priority(2)=1
visited(1)=true
</State 1 (controlled)>
Insert a constant in Node of type Integer for frontier:
2
Insert a set {} of constant in Node of type Integer for neighbors(2):
Insert a set:
{3}
Insert a integer constant for distance(2,3):
1
Insert a integer constant for heuristic(3,9):
0
<State 2 (controlled)>
costSoFar(1)=0
costSoFar(2)=1
costSoFar(3)=2
currentNode=2
first_run=false
previous(1)=1
previous(2)=1
previous(3)=2
priority(1)=undef
priority(2)=undef
priority(3)=2
visited(1)=true
visited(2)=true
</State 2 (controlled)>
Insert a constant in Node of type Integer for frontier:
3
Insert a set {} of constant in Node of type Integer for neighbors(3):
```

```
Insert a set:
{4}
Insert a integer constant for distance(3,4):
1
Insert a integer constant for heuristic(4,9):
0
<State 3 (controlled)>
costSoFar(1)=0
costSoFar(2)=1
costSoFar(3)=2
costSoFar(4)=3
currentNode=3
first_run=false
previous(1)=1
previous(2)=1
previous(3)=2
previous(4)=3
priority(1)=undef
priority(2)=undef
priority(3)=undef
priority(4)=3
visited(1)=true
visited(2)=true
visited(3)=true
</State 3 (controlled)>
Insert a constant in Node of type Integer for frontier:
4
Insert a set {} of constant in Node of type Integer for neighbors(4):
Insert a set:
{5, 6}
Insert a integer constant for distance(4,5):
1
Insert a integer constant for heuristic(5,9):
3
Insert a integer constant for distance(4,6):
3
Insert a integer constant for heuristic(6,9):
0
<State 4 (controlled)>
costSoFar(1)=0
costSoFar(2)=1
costSoFar(3)=2
costSoFar(4)=3
costSoFar(5)=4
costSoFar(6)=6
currentNode=4
first_run=false
previous(1)=1
previous(2)=1
previous(3)=2
```

```
previous(4)=3
previous(5)=4
previous(6)=4
priority(1)=undef
priority(2)=undef
priority(3)=undef
priority(4)=undef
priority(5)=7
priority(6)=6
visited(1)=true
visited(2)=true
visited(3)=true
visited(4)=true
</State 4 (controlled)>
Insert a constant in Node of type Integer for frontier:
5
Insert a set {} of constant in Node of type Integer for neighbors(5):
Insert a set:
{6}
Insert a integer constant for distance(5,6):
-1
Insert a integer constant for heuristic(6,9):
0
<State 5 (controlled)>
costSoFar(1)=0
costSoFar(2)=1
costSoFar(3)=2
costSoFar(4)=3
costSoFar(5)=4
costSoFar(6)=3
currentNode=5
first_run=false
previous(1)=1
previous(2)=1
previous(3)=2
previous(4)=3
previous(5)=4
previous(6)=5
priority(1)=undef
priority(2)=undef
priority(3)=undef
priority(4)=undef
priority(5)=undef
priority(6)=3
visited(1)=true
visited(2)=true
visited(3)=true
visited(4)=true
visited(5)=true
</State 5 (controlled)>
```

```
Insert a constant in Node of type Integer for frontier:
6
Insert a set {} of constant in Node of type Integer for neighbors(6):
Insert a set:
{7}
Insert a integer constant for distance(6,7):
1
Insert a integer constant for heuristic(7,9):
0
<State 6 (controlled)>
costSoFar(1)=0
costSoFar(2)=1
costSoFar(3)=2
costSoFar(4)=3
costSoFar(5)=4
costSoFar(6)=3
costSoFar(7)=4
currentNode=6
first_run=false
previous(1)=1
previous(2)=1
previous(3)=2
previous(4)=3
previous(5)=4
previous(6)=5
previous(7)=6
priority(1)=undef
priority(2)=undef
priority(3)=undef
priority(4)=undef
priority(5)=undef
priority(6)=undef
priority(7)=4
visited(1)=true
visited(2)=true
visited(3)=true
visited(4)=true
visited(5)=true
visited(6)=true
</State 6 (controlled)>
Insert a constant in Node of type Integer for frontier:
7
Insert a set {} of constant in Node of type Integer for neighbors(7):
Insert a set:

Expected { but found EOF
1
Expected { but found 1
0
Expected { but found 0
```

```
{8}
Insert a integer constant for distance(7,8):
1
Insert a integer constant for heuristic(8,9):
0
<State 7 (controlled)>
costSoFar(1)=0
costSoFar(2)=1
costSoFar(3)=2
costSoFar(4)=3
costSoFar(5)=4
costSoFar(6)=3
costSoFar(7)=4
costSoFar(8)=5
currentNode=7
first_run=false
previous(1)=1
previous(2)=1
previous(3)=2
previous(4)=3
previous(5)=4
previous(6)=5
previous(7)=6
previous(8)=7
priority(1)=undef
priority(2)=undef
priority(3)=undef
priority(4)=undef
priority(5)=undef
priority(6)=undef
priority(7)=undef
priority(8)=5
visited(1)=true
visited(2)=true
visited(3)=true
visited(4)=true
visited(5)=true
visited(6)=true
visited(7)=true
</State 7 (controlled)>
Insert a constant in Node of type Integer for frontier:
8
Insert a set {} of constant in Node of type Integer for neighbors(8):
Insert a set:
{9}
Insert a integer constant for distance(8,9):
1
Insert a integer constant for heuristic(9,9):
0
<State 8 (controlled)>
```

```
costSoFar(1)=0
costSoFar(2)=1
costSoFar(3)=2
costSoFar(4)=3
costSoFar(5)=4
costSoFar(6)=3
costSoFar(7)=4
costSoFar(8)=5
costSoFar(9)=6
currentNode=8
first_run=false
previous(1)=1
previous(2)=1
previous(3)=2
previous(4)=3
previous(5)=4
previous(6)=5
previous(7)=6
previous(8)=7
previous(9)=8
priority(1)=undef
priority(2)=undef
priority(3)=undef
priority(4)=undef
priority(5)=undef
priority(6)=undef
priority(7)=undef
priority(8)=undef
priority(9)=6
visited(1)=true
visited(2)=true
visited(3)=true
visited(4)=true
visited(5)=true
visited(6)=true
visited(7)=true
visited(8)=true
</State 8 (controlled)>
Insert a constant in Node of type Integer for frontier:
9
<State 9 (controlled)>
costSoFar(1)=0
costSoFar(2)=1
costSoFar(3)=2
costSoFar(4)=3
costSoFar(5)=4
costSoFar(6)=3
costSoFar(7)=4
costSoFar(8)=5
costSoFar(9)=6
```

```
currentNode=9
first_run=false
previous(1)=1
previous(2)=1
previous(3)=2
previous(4)=3
previous(5)=4
previous(6)=5
previous(7)=6
previous(8)=7
previous(9)=8
priority(1)=undef
priority(2)=undef
priority(3)=undef
priority(4)=undef
priority(5)=undef
priority(6)=undef
priority(7)=undef
priority(8)=undef
priority(9)=undef
visited(1)=true
visited(2)=true
visited(3)=true
visited(4)=true
visited(5)=true
visited(6)=true
visited(7)=true
visited(8)=true
visited(9)=true
</State 9 (controlled)>
Insert a constant in Node of type Integer for frontier:
9
<State 10 (controlled)>
costSoFar(1)=0
costSoFar(2)=1
costSoFar(3)=2
costSoFar(4)=3
costSoFar(5)=4
costSoFar(6)=3
costSoFar(7)=4
costSoFar(8)=5
costSoFar(9)=6
currentNode=9
first_run=false
previous(1)=1
previous(2)=1
previous(3)=2
previous(4)=3
previous(5)=4
previous(6)=5
```

```
previous(7)=6
previous(8)=7
previous(9)=9
priority(1)=undef
priority(2)=undef
priority(3)=undef
priority(4)=undef
priority(5)=undef
priority(6)=undef
priority(7)=undef
priority(8)=undef
priority(9)=undef
visited(1)=true
visited(2)=true
visited(3)=true
visited(4)=true
visited(5)=true
visited(6)=true
visited(7)=true
visited(8)=true
visited(9)=true
</State 10 (controlled)>
Final state:
costSoFar(1)=0
costSoFar(2)=1
costSoFar(3)=2
costSoFar(4)=3
costSoFar(5)=4
costSoFar(6)=3
costSoFar(7)=4
costSoFar(8)=5
costSoFar(9)=6
currentNode=9
first_run=false
previous(1)=1
previous(2)=1
previous(3)=2
previous(4)=3
previous(5)=4
previous(6)=5
previous(7)=6
previous(8)=7
previous(9)=9
priority(1)=undef
priority(2)=undef
priority(3)=undef
priority(4)=undef
priority(5)=undef
priority(6)=undef
priority(7)=undef
```

```
priority(8)=undef
priority(9)=undef
visited(1)=true
visited(2)=true
visited(3)=true
visited(4)=true
visited(5)=true
visited(6)=true
visited(7)=true
visited(8)=true
visited(9)=true
```