Vrije Universiteit Brussel
Faculteit van de Wetenschappen
Departement Informatica

Reflectie door middel van Jit-compilatie

Proefschrift ingediend met het oog op het behalen van de graad van licentiaat in de informatica

Door : Paul-Henri Van der Steichel
Promotor : Prof. Dr. Theo D'Hondt
Begeleider : Karsten Verelst

Academiejaar 2001-2002

Samenvatting

Heden ten dage is reflectiviteit weer een hot topic in de informatica sector. Om een performant reflectief systeem te verkrijgen, waarbij interpretatie het sleutelwoord is, moet men snelheid afwegen tegenover reflectie. Just-in-Time compilatie is een techniek die de voordelen van compilatie en interpretatie samenbundelt. In deze thesis wordt Jit-compilatie (just-in-time) bekeken als basis van een reflectieve virtuele machine.

Inhoudsopgave

1 Introductie **7**
1.1 Doel 7
1.2 Reisgids 8
1.3 Dankbetuiging 8
1.4 Historische commentaar[19] 9

I Huidige technologien **10**

2 De RVM **10**
2.1 Inleiding 10
2.2 Definities 10
2.3 Categorizeren van Reflectie 11
2.4 De Reflectieve Virtuele Machine 14
2.5 De oneindige toren van interpreters 15
2.6 Interpretatie en Compilatie 16
2.7 Conclusie 17

3 Pico/Borg **18**
3.1 Pico 18
3.2 `Borg` 19
3.3 Borg Virtuele Machine 19
3.4 Borg Computational Model 21
3.5 Borg intern 23
3.5.1 Algemeen 23
3.5.2 Continuaties 23
3.6 Strong Mobility 29
3.7 Conclusie 31

4 Jit-compilatie **34**
4.1 inleiding 34
4.2 Java & JIT-compilatie 35
4.3 Java Jit compiler overzicht 35
4.4 Conclusie 38

II Huidige RVM's **39**

5 RbCl **39**
5.1 Kernel-less systeem 41
5.2 Design en implementatie van het RbCl Metasysteem 42
5.2.1 Linguistic symbiosis 42
5.2.2 Metasysteem 43
5.2.3 De oneindige toren van directe implementaties. 44
5.2.4 Level Shifting door level managers 47
5.2.5 Implementatie van de Linguistic Symbiosis 48
5.2.6 Generatie van het meta meta systeem 48
5.3 Conclusie 49

6 Virtual Virtual Machine **51**
6.1 De VVM Architectuur 51
6.2 VVM Werking 53
6.3 Conclusie 53

7 NitrO **55**
7.1 Overzicht 55
7.2 Non-restrictive Computational Reflection System 56
7.3 conclusie 58

8 Black **59**
8.1 Inleiding 59
8.2 De oneindige toren 59
8.3 Implementatie Moeilijkheden 60
8.4 Performante Interpretatie 60
8.5 De structuur van de Black interpreter 61
8.6 Construeren van de default I_n 62
8.7 Compileren d.m.v. een partiele evaluator 62

9 Conclusie **63**

III Reflectie door middel van Jit-compilatie **65**

10 De RVM met Jit-compilatie **65**
10.1 Inleiding 65
10.2 Design 66
10.3 Problemen 66
10.4 Versie- / Dependency systeem 68
10.5 SMART-Jit 69
10.6 Linguistic Symbiosis 70

10.7 Wanneer en wat Jit-compilen? 71
10.8 Dynamisch linken[21] 72
10.9 Werking 72

11 Conclusie **74**

12 Future work **75**

Lijst van figuren

1 reflectie en reificatie in een computational systeem 12
2 compile-time meta architectuur 13
3 Load-time meta architectuur 13
4 Run-time meta architectuur 13
5 Boom voorstelling van 'plus(a,b):{a+b}; 15
6 boom van (5 + 4) * 3 24
7 MUL continuatie uit NatArithmetic.c 24
8 BINARY macro uit Natives.h 24
9 BINARY_BODY macro uit Natives.h 25
10 EXP continuatie uit Evaluator.c 26
11 APL continuatie uit Evaluator.c 27
12 NAT continuatie uit Evaluator.c 27
13 ADD continuatie uit NatArithmetic.c 28
14 UNABINARY macro uit Natives.h 28
15 AD2 continuatie uit NatArithmetic.c 30
16 SWP continuatie uit Natives.c 31
17 MU1 continuatie uit NatArithmetic.c 32
18 Een overzicht van de Jit-compiler 37
19 Een systeem met en zonder een runtime kernel 41
20 De oneindige toren van directe implementatie 46
21 De VVM architectuur 52
22 Development process 56
23 Free-context grammar rule 57
24 De structuur van de 2 levels in generische interpretatie 58
25 Overzicht van Black 61
26 Design van Reflectieve Virtuele Machine 66

1 Introductie

> To steal ideas from one person is plagiarism; to steal from many is research – **Anonymous**

Vandaag staan de meeste computers wel verbonden in een netwerk. Is het niet de gehele tijd, dan toch af en toe via een dial-up lijn met het internet bijvoorbeeld. Het samen gebruiken van resources en data wordt in deze tijd belangrijker en belangrijker. Men noemt deze tijd niet voor niets de informatie-maatschappij. Met deze integratie van programma's en data komen natuurlijk ook een aantal problemen mee. Niet iedereen gebruikt compatible hardware en software.

Java was reeds een stap in de goede richting. Er werd een virtuele machine gedefinieerd die op alle platvormen kon geimplementeerd worden, en de gebruikers-programma's werden op deze virtuele machines uitgevoerd. Dit wil zeggen dat alle Java programma's op alle (goed geconstrueerde) Java Virtuele Machines konden uitgevoerd worden.

Met deze technologie kwamen ook de agentsystemen tevoorschijn. Kleine autonome programma's die zich op het netwerk van platvorm naar platvorm begaven en taken voor hun gebruiker uitvoeren. Met deze agents, had men ook nood aan sterke mobiliteit. Sterke mobiliteit wil zeggen dat de agent midden in zijn bewerking kon stopgezet worden. De reeds uitgevoerde bewerking bewaren en meeverhuizen naar een andere machine, en daar zijn uitwerking verder zetten. Hiervoor had men reflectie nodig. Reflectie wil op zijn beurt zeggen dat het programma + uitvoering expliciet worden gemaakt. De uitwerking van programma's gebeurt niet meer ergens ver weg waar niemand aan kan, maar de gebruiker kan tijdens de uitwerking deze interactief gaan wijzigen en/of vastnemen en bewaren. Het geheim achter reflectie noemt: de oneindige toren van interpreters. Er wordt een oneindige toren van interpreters gecreeerd waarbij elke interpreter geinterpreteerd wordt door een interpreter 1 laag hoger. Daar elke interpreter geinterpreteerd wordt, is het mogelijk elke interpreter te veranderen.

1.1 Doel

In deze thesis willen we testen of we de oneindige toren van interpreters in een reflectieve talen kunnen terugbrengen tot 1 interpreter door middel van Jit-compilatie. Hiervoor zullen we met een nieuw design op de proppen moeten komen, want Jit-compilatie en reflectie gaan niet goed samen. Tot op heden wordt reflectie ondersteund door (een oneindige toren van) interpreters die het programma interpreteren. Want eenmaal een programma gecompileerd

is, ligt zijn gedrag vast, en kan daar niets meer aan veranderd worden. Als we dan toch het gedrag willen aanpassen is opnieuw compileren de boodschap.

1.2 Reisgids

Deze thesis is opgebouwd uit 3 delen. In een eerste deel bespreken we een aantal bestaande technologien die we nodig hebben in deze thesis. We zien in hoofdstuk 2 de definitie van de Reflectieve Virtuele Machine (RVM). Hierin trachten we eerst reflectie uit te leggen waarna we uitleggen hoe we reflectie verkrijgen. In hoofdstuk 3 zien we dan Pico/Borg. Borg is een Reflectieve Virtuele Machine. We zien in dit hoofdstuk wat Borg is en hoe het werkt. In hoofdstuk 4 gaan we dan dieper in op Just-in-time compilatie. Jit-compilatie is de technologie die we gaan gebruiken om de reflectieve machine performanter te maken.

In deel 2 gaan we bekijken hoe de huidige Reflectieve Virtuele Machines opgebouwd zijn. We gaan hun voordelen en nadelen onderzoeken. We doen dit aan de hand van RbCl, VVM, NitrO en Black.

In deel 3 gaan we dan een Reflectieve Virtuele Machine definieren die gebruik maakt van een jit-compiler. Deze virtuele machine zal at runtime aanpasbaar zijn, en toch zal dit niet lijden onder performantie verlies. Hierbij proberen we ook de voordelen die we in onze literatuursstudie zijn tegen gekomen te integreren in deze RVM.

Deze thesis is niet in perfect algemeen Nederlands geschreven omdat sommige termen onvertaalbaar zijn zonder een onverstaanbare tekst over te houden. Waar we konden hebben we Nederlands op een consistente wijze gehanteerd. Vaktermen en code hebben we in hun standaartaal, het Engels, gelaten.

1.3 Dankbetuiging

Vooraleer we van start gaan zou ik eerst nog een aantal mensen willen bedanken voor hun hulp bij het schrijven van deze thesis :

- Prof. dr. Theo D'Hondt, de promotor van deze thesis
- Karsten Verelst, mijn begeleider bij deze thesis. Hij heeft mij altijd bijgestaan met raad en daad.

Ook zou ik een aantal personen willen bedanken die mijn thesis hebben nagelezen en mij gewezen hebben op de schrijffouten

- Mijn ouders, mijn broer

Ook zou ik de mensen van het PROG lab willen bedanken voor de tips en opmerkingen tijdens de wekelijkse thesis vergaderingen

1.4 Historische commentaar[19]

rond 1945 :

De computer architectuur, uitgevonden door John von Neumann bewaart code en data in hetzelfde geheugen. Dit maakt het mogelijk om code als data te beschouwen. Het eerste gebruik van deze eigenschap was dynamisch linken, het aanpassen van 'jumps' bij het verplaatsen van code. Aanpassingen van programma's door zichzelf ("self modifying code") werd snel gezien als slechte programmeerkunde, gedeeltelijk omdat er geen formeel model was gevonden als basis voor zulke code aanpassingen.

1972..1980 :

Smalltalk werd ontwikkeld. Smalltalk maakt gebruik van meta klassen die informatie bijhouden over klassen en behandelt klassen als first-class objecten die instanties zijn van een meta klasse.

1981 :

Men ontdekt dat men performantie winst heeft als men het virtuele geheugen sub-systeem hints geeft over het geplande geheugen gebruik in een Smalltalk mark-sweep garbage collector.

1982 :

Brian C. Smith ontwikkelt 3-Lisp, een reflectief dialect van Lisp. 3-Lisp is de eerste taal die reflectie als een belangrijk taal concept beschouwt.

1984 :

Men scheidt het concept van reificatie en het concept van de toren van interpreters.

1987 :

Pattie Maes ontwikkelt 3-KRS om te demonstreren dat het mogelijk is om een reflectieve object georienteerde taal te construeren.

1991 :

Kiczales, Rivieres and Bobrow ontwikkelen het CLOS Metaobject protocol.

2001 :

Theo D'Hondt, Werner Van Belle en Karsten Verelst definieren de Reflectieve Virtuele Machine

Deel I
Huidige technologien

2 De RVM

> To read without reflecting is like eating without digesting
> – **Edmund Burke**

2.1 Inleiding

Als we de taalsemantiek van een conventionele niet-reflectieve taal willen aanpassen, moeten we de interpreter, of de compiler, van deze taal aanpassen. Deze compiler/interpreter is meestal geschreven in low level code waarbij we implementatie details goed moeten kennen als we deze willen aanpassen.

Reflectieve talen daarentegen laten ons toe om in hetzelfde taal framework aanpassingen te maken aan de interpreter/compiler op een hoger level niveau zonder rekening te hoeven houden met implementatie details.

De laatste tijd zijn nogal vele reflectie technieken ontworpen om aanpasbare systemen te ontwikkelen. Compile time reflectie laat toe om aanpasbare programmeertalen te ontwikkelen met efficiente performantie (bijv. openC++, MPC++ of openJava). Maar deze zijn niet aanpasbaar at runtime. Aan de andere kant zijn de meeste runtime-reflectie systemen dan weer gebaseerd op de mogelijkheid tot het aanpassen van de taalsemantiek tijdens de uitvoering. Deze aanpasbaarheid wordt bereikt door het implementeren van een protocol. Dit protocol specifieert dan wat wel en wat niet mag aangepast worden : het MetaObject Protocol of kortweg MOP.

2.2 Definities

We introduceren eerst een paar termen.

Def.1 Reflectie

Reflection is the ability of a program to manipulate as data, something representing the state of the program during its own execution. There are two aspects of such manipulation : introspection and intercession. Introspection is the ability of the program to observe and therefore reason about its own state. Intercession is the ability of a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a

mechanism for encoding execution state as data; providing such an encoding is called reification[?]

Def.2 Meta-level object

Een meta-level object is een object dat zich in het meta-level bevindt en dat meta-level functionaliteiten implementeert.[19]

Def.3 Metaobject

Een metaobject is een object dat informatie bevat over andere objecten (de base-level objecten) en/of ook de uitvoering van het object controleert.[19]

Def.4 Meta-circulaire interpreter

Een interpreter die geschreven is in de taal die het interpreteert (host taal), gebruik makende van de faciliteiten van de host taal.[19]

Def.5 Reificatie

Reificatie is het toegankelijk maken van iets wat normaal gesproken niet voor handen is in het programmeermodel of verborgen werd voor de programmeur.[19]

Def.6 Basis systeem

Het basis systeem is het computationeel systeem hetwelke het domein is voor een ander computationeel systeem, nl. het meta systeem.[19]

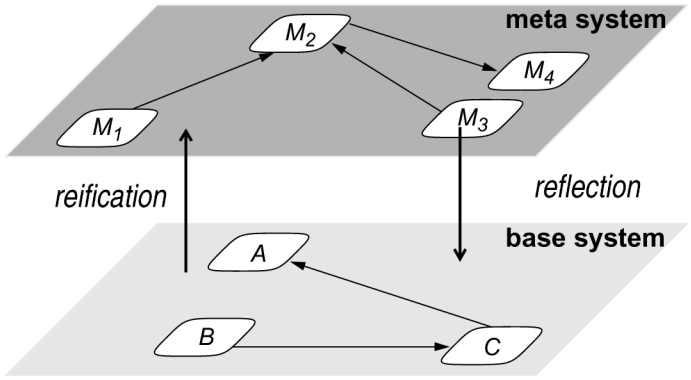
Def.7 Meta systeem

Het meta systeem is het computationeel systeem hetwelke het domein is voor een ander computationeel systeem, nl. het basis systeem. [19]

2.3 Categorizeren van Reflectie

We identificeren 2 hoofdcriteria als we reflectieve systemen willen categorizeren. Deze criteria zijn **wanneer** reflectie plaats vindt en **wat** gereflecteerd wordt. Op 'wat' gereflecteerd wordt kunnen we volgend onderscheid maken :

- **Introspectie.** Het systeem kan 'bekeken' worden, maar er kan niets gewijzigd worden. Als we Java als voorbeeld nemen, dan kunnen we de java.lang.reflect package hieronder classificeren. We kunnen informatie over de klassen objecten, methodes en velden at runtime verkrijgen

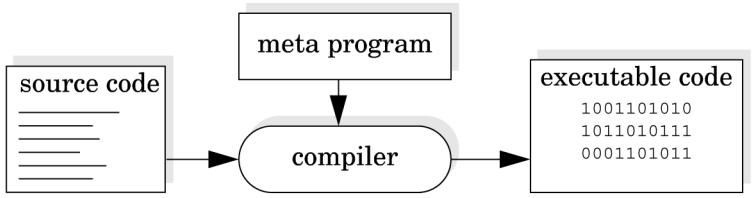


Figuur 1: reflectie en reificatie in een computational systeem

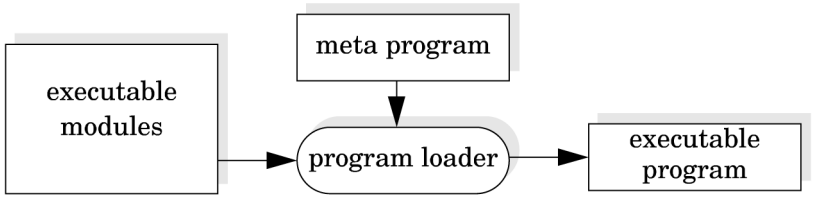
- **Structurele reflectie.** De structuur van het systeem kan dynamsich gewijzigd worden. Een voorbeeld van deze reflectie is het toevoegen van een veld aan een object.
- **Computationele Reflectie.** Het gedrag van het systeem kan aangepast worden. In de API v.1.3 van Java werd de java.lang.reflect.Proxy package toegevoegd. Deze kan gebruikt worden om het dispatching-methode-mechanisme, gebruikt door een proxy object, te veranderen.
- **Linguistic Reflection.** De (reflectieve) programmeertaal heeft de mogelijkheid om zichzelf aan te passen, bijv. het aanpassen van zijn eigen lexicale of syntactische specificaties. Als voorbeeld nemen we bijvoorbeeld OpenJava. Deze taal kan aangepast worden aan specifieke design patterns.

Als we kijken naar 'wanneer' reflectie plaats heeft, kunnen we volgend onderscheid maken :

- **Compile-time reflectie.** Het aanpassen van het systeem vindt plaats at compile-time (bijv. openJava). De voordelen van dit systeem zijn 1) een betere runtime performantie en 2) de mogelijkheid om zijn eigen taal aan te passen (linguistic reflection). Het nadeel is dan wel dat het systeem niet meer volledig flexibel is. (zie figuur 2)
- **Load-time reflectie.** Load-time reflectie gebeurt dan tijdens het laden en linken van een programma, juist voor dat het uitgevoerd



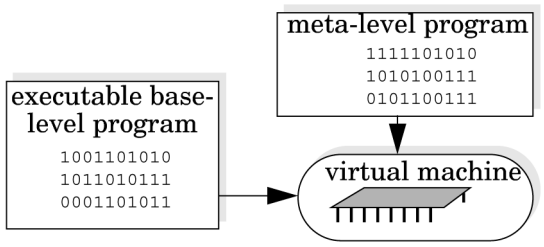
Figuur 2: compile-time meta architectuur



Figuur 3: Load-time meta architectuur

wordt. Het OMOS systeem is een voorbeeld van een load-time reflectie systeem. Wanneer er gebruik wordt gemaakt van load-time reflectie is er enkel overhead bij het opstarten van het systeem. Het nadeel is dan weer dat er geen mogelijkheid is van at runtime aanpasbaarheid. (zie figuur 3)

- **Runtime Reflectie.** Het systeem kan at runtime aangepast worden (bijv.metaX, vroeger bekend als MetaJava). Deze systemen zijn zeer aanpasbaar, maar zijn wel veel trager. Computationele reflectieve systemen zijn meestal geimplementeerd gebruik makende van runtime reflectie, maar zij bezitten geen linguistic reflectie. (zie figuur 4)



Figuur 4: Run-time meta architectuur

2.4 De Reflectieve Virtuele Machine

Na deze categorisatie van reflectie gaan we een definitie geven van de Reflectieve Virtuele Machine. De Reflectieve Virtuele Machine is een systeem dat toelaat om elke feature van de programmeertaal te wijzigen at runtime en dit zonder enige restrictie opgelegd door een protocol.

We definieren een reflectieve virtuele machine als een virtuele machine die zijn volledige computationele staat kan vatten, inclusief zijn abstracte grammatica, het geheugenmodel, de stack en zijn primitieven.

Zoals hierboven reeds vermeld, is een RVM een machine die zo veel mogelijk reflectie geeft aan de gebruiker als mogelijk is. Dit start met de reflectie van de abstracte grammatica. Wanneer een programmeur een programma in de virtuele machine brengt, dan wordt deze geparsed en omgevormd tot een boom-structuur, de abstracte grammatica. Onder reflectie van de abstracte grammatica verstaan we dat de knopen van de boom expliciet worden gemaakt in de taal en dat we dus rechtstreeks op deze boom kunnen werken en knopen veranderen naar wens. We nemen het voor de hand liggende voorbeeld van het optellen van 2 nummers. Dit programma kan beschreven worden in Scheme als

(cons '+ '(1 2))

We kunnen dus programma's gaan voorstellen als gewone data, en we kunnen hen ook op deze manier gebruiken. Het idee dat hieruit volgt is dat een programma zichzelf als data kan zien, en dus zichzelf kan aanpassen.

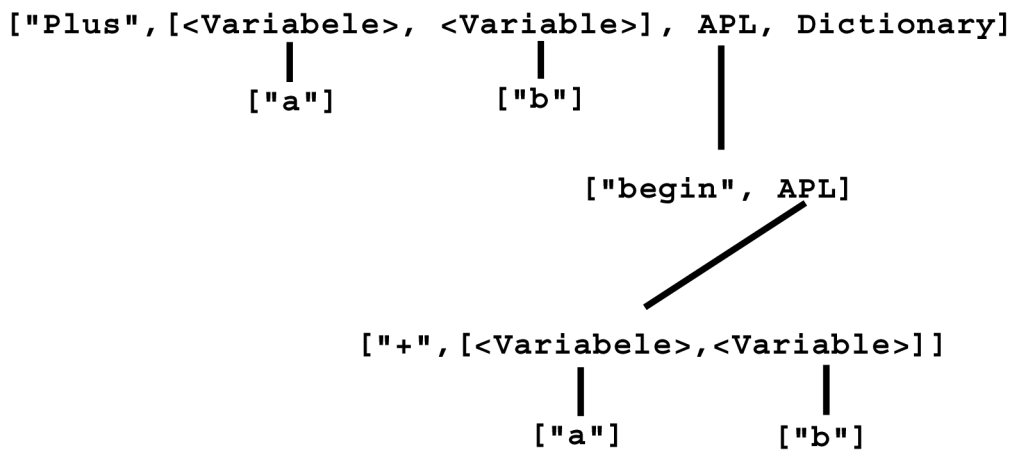
Bijv.

plus(a,b):{a+b}

Dit programma contrueert een boom zoals in figuur 5. In deze boom kunnen we in de body "a" vervangen door 5 en zo het gedrag van plus veranderen.

Een soortgelijk voorbeeld kan gevonden worden in de java.lang.reflect package. Alle methoden die in deze package zitten geven toegang tot de interne representatie van de Java 'first class' data structuren, dit wel zonder de mogelijkheid om deze te wijzigen, enkel observeren. De meeste populaire programmeertalen bezitten deze eigenschap. Reflectie van de abstracte grammatica is hetzelfde als reflectie van de code.

Vervolgens hebben we de reflectie van de stack. Dit wil zeggen dat de meta-level stack expliciet moet zijn en liefst opgeslagen in een object level data structuur. In de praktijk wil dit zeggen dat de stack van de virtuele machine moet gecreeerd worden in de heap van de interpreter, en het best geimplementeerd is als een soort tabel of lijst. Nog anders gezegd moeten we de stack kunnen behandelen als een object in onze geinterpreteerde taal en



Figuur 5: Boom voorstelling van 'plus(a,b):{a+b};

niet alleen in de taal waarin de interpreter geschreven is.

Tenslotte moet ook de omgeving (environment) gereflecteerd worden.

Als deze drie metalevel datastructuren gereflecteerd worden in het object level, dan kunnen we de volledige computationele staat vatten. En het vatten van deze computationele staat (voor sterke mobiliteit) was het doel om een RVM te creeeren.

2.5 De oneindige toren van interpreters

Een belangrijke eigenschap van de reflectieve virtuele machines is hun toren structuur. In reflectieve talen worden de gebruikersprogramma's op base-level (of level 0) geinterpreteerd door een interpreter die 1 level hoger staat (level1 of metalevel), dewelke op zijn beurt geinterpreteerd wordt door een interpreter die runt op level 2 en zo voort. De reden dat we zo een toren van interpreters nodig hebben is dat de interpreter op level 1 aanpasbaar hoort te zijn. Deze aanpassingen moeten geinterpreteerd worden. Daarvoor dient dan de interpreter op level 2. Deze interpreter op level 2 moet op zijn beurt ook aanpasbaar zijn dus dient er ook een interpreter op level 3 aanwezig te zijn om de interpreter op level 2 te interpreteren. Conceptueel is er zo een oneindige toren van interpreters. Reflectieve talen laten gebruikersprogramma's toe om programma's uit te voeren op eender welk level in deze toren en geven de mogelijkheid om toegang te krijgen (introspecting) of het veranderen van interpreters van de bovenste levels.

Deze oneindige toren van interpreters is niet te implementeren met een

eindig aantal resources omdat elke interpreter geinterpreteerd wordt door een interpreter 1 level hoger en veroorzaakt zo een oneindige regressie. Een oplossing hiervoor werd gegeven door Jefferson en Friedman door het limiteren van de oneindige interpreters tot een aantal levels waarbij we omhoog kunnen gaan. Een probleem van deze approach is dat elke interpreter moet geinterpreteerd worden door een interpreter 1 level hoger en dit maakt het geheel extreem traag.

Een andere mogelijkheid om deze oneindige toren efficient te implementeren is door het creeeren van direct uitgevoerde interpreters op een lazy manier. Deze methode laat ons wel niet toe om de interpreters zelf te herdefinieren omdat zij direct uitgevoerd worden in machine taal. We kunnen wel vrij op en neer gaan in de toren van interpreters en de omgevingen, continuaties enzovoort veranderen, maar de interpreters zelf zijn onwijzigbaar.

2.6 Interpretatie en Compilatie

Interpretatie d.m.v. de oneindige toren van interpreters is het keywoord in reflectieve virtuele machines. We zullen even dieper ingaan op het verschil tussen interpretatie en compilatie. Veronderstel een programma P geschreven in Scheme. Als P geinterpreteerd wordt door een Scheme interpreter I zoals in I(P,env) dan kunnen we delen van P veranderen. Bijvoorbeeld als we de functie f in P willen veranderen in g is 'set! f g' het enige dat we moeten doen. Na dit is de waarde van f in de environment veranderd naar g. Omdat de omgeving elke keer bij een functie oproep wordt nagekeken wordt g uitgevoerd wanneer f wordt opgeroepen.

We kunnen ook meer gesofistikeerde veranderingen aanbrengen door gebruik te maken van de reflectieve faciliteiten. Veronderstel dat we de volgorde van evaluatie van argumenten willen veranderen. Dit wordt meestal beschreven in eval-list (of evlist) in I. Dus als we eval-list veranderen door een nieuwe my-eval-list gedefinieerd door de gebruiker, gebruik makende van de reflectieve faciliteiten dan zal de evaluatie volgorde veranderd zijn door diegene gedefinieerd door my-eval-list. Omdat P geinterpreteerd wordt door I kunnen we het gedrag van P aanpassen als we veranderingen mogen aanbrengen aan I. Dit kan omdat de interpreter I alle informatie heeft hoe P moet geinterpreteerd worden.

Wat als we P nu compileren in machine code P en deze direct uitvoeren. Dan zullen alle veranderingen die hierboven beschreven zijn niet meer mogelijk zijn omdat P direct uitgevoerd wordt. Zelfs al zouden we de uitvoering stopzetten en f veranderen in g, dan nog zou de uitvoering niet veranderen omdat P gecompileerd is en geen opzoeking in de omgeving meer doet. De herdefinitie van eval-list zou ook het gedrag van P niet meer beinvloeden om-

dat *P* niet meer geinterpreteerd wordt door I. Compilatie heeft dus een hoge efficientie omdat het er van uit gaat dat het programma niet meer veranderd wordt. Als we dus de veranderingen die hierboven beschreven staan willen toelaten moet P dus expliciet geinterpreteerd worden door gebruik te maken van I.

2.7 Conclusie

In dit hoofdstuk zijn we dieper ingegaan op het begrip reflectiviteit (in al zijn vormen) en in het bijzonder op de reflectieve virtuele machine.

De reflectieve virtuele machine werd ontwikkelt om een at runtime aanpasbaar systeem te verkrijgen dat zijn computationele staat kon vatten. Om deze computationele staat te kunnen vatten moeten we de abstracte grammatica, de stack en de environment kunnen vatten. Hiervoor is reflectie nodig. Reflectie steunt op het principe van de oneindige toren van interpreters waarbij elke interpreter een interpreter interpreteert die zich 1 level hoger in de toren bevindt. Op die manier is het mogelijk om de interpreters te veranderen daar deze veranderingen geinterpreteerd wordt door een interpreter 1 level hoger.

Indien compilatie zou gebruikt worden, dan werd het programma niet meer geinterpreteerd maar direct uitgevoerd. Wat wil zeggen dat er geen opzoekingen meer worden gedaan in het environment en dat noch de interpreter tussen komt om de uitvoering te regelen. Compilatie is een techniek die er van uitgaat dat het programma niet meer verandert en hierdoor een performante onaanpasbare representatie maakt. Als we dus at runtime aanpasbare systemen willen hebben, is interpretatie het sleutelwoord.

3 Pico/Borg

> Silence is foolish if we are wise, but wise if we are foolish –
> **Charles Caleb Colton**

In dit hoofdstuk bespreken we Pico/Borg want deze programmeertaal is het uitgangspunt voor onze reflectieve virtuele machine met Jit-compilatie. Niet alleen zal een kleine inleiding in deze taal gegeven worden, ook zullen we dieper ingaan op de interpretatie van programma's geschreven in Pico. In subsectie 3.1 geven we een kort overzicht van het waarom van Pico. Daarna komt u meer te weten over Borg. Daarbij geven we eerst een algemene situering, wat Borg nu eigenlijk is en waar het verschilt met Pico. Daarna doen we uit de doeken uit welke delen Borg bestaat en hoe het computational model in elkaar zit om dan een interpretatie van een programma te volgen. Uiteindelijk besluiten we dit hoofdstuk door nog wat over sterke mobiliteit (strong mobility) te vertellen.

3.1 Pico

Pico is een kleine programmeertaal ontwikkeld aan de Vrije Universiteit Brussel. Deze taal werd gecreeerd om niet-informatica studenten een basiscursus informatica te onderwijzen. De nadruk in deze programmeertaal ligt dus op eenvoud en consistentie van de syntax. Uitzonderingen en moeilijke constructies zijn uit den boze. Toch is de expressiviteit zeer groot, vergelijkbaar met Scheme [17]. De Pico semantiek is gedefinieerd door een verzameling van negen evaluatie-functies die ondersteund worden door een geheugenmodel en een computationeel model. Het geheugenmodel ondersteunt volledig opslagmanagement en heropeising; het computationeel model is gebaseerd op een 'push-down' automaat die de expressies en continuaties managed op een dubbele stack. Continuaties, geinspireerd door CPS (continuation passing style), zijn thunks die in volgorde worden gezet om zo de uitwerking te ondersteunen. Pico heeft minder dan 20 continuaties nodig om zijn volledige semantiek te implementeren.

Een klein voorbeeld van Pico-code :

```
Hello-World() :
{
display("Hello World")
}
:<function Hello-World>
```

3.2 Borg

Borg is het gedistribueerde broertje van Pico, en is een mobiel multi-agent platvorm. We kunnen dus agents creeeren die zich op het platvorm verplaatsen en taken uitvoeren.

> **Mobile Agents** are autonomous, intelligent programs that move through a network, searching for and interacting with services on the user's behalf. These systems use specialized servers to interpret the agent's behaviour and communicate with other servers. A Mobile Agent has inherent navigational autonomy and can ask to be sent to some other nodes. Mobile Agents should be able to execute on every machine on a network and the agent code should not have to be installed on every machine the agent could visit. Therefore Mobile Agents use mobile code systems like Java and the Java virtual machine where classes can be loaded at runtime over the network.[4]

Een voorbeeld van een agent-systeem kan bijvoorbeeld een gedistribueerde agenda zijn. We creeeren een agent met de opdracht om een afspraak te maken met X, Y en Z. Onze agent verplaatst zich op het netwerk op zoek naar X, Y en Z en zoekt naar een mogelijke vrij tijdsinterval waarbij de 4 personen een afspraak kunnen maken.

Borg presenteert een architectuur die tot doel heeft tot het programmeren van mobiele, intelligente, autonome agenten, waarbij de programmeur geen rekening hoeft te houden met allerlei lastig te programmeren zaken, zoals 'strong mobility'[1], manier van communicatie tussen agenten, garbage collection e.a.

3.3 Borg Virtuele Machine

De Borg virtuele machine bestaat uit :

- **Een taal processor** dewelke een abstracte grammatica genereert van een gegeven programma-tekst. In Borg bijvoorbeeld, zal het programma

  set(nr):: r := nr

  vertaald worden in de volgende expressie :

  [DCL[APL set [TAB [REF nr]]][ASS r [REF nr]]]

[1] Strong mobility is dat een agent zichzelf zelf tijdens een uitvoering van een actie, deze actie kan stoppen, verplaatsen naar een andere plaats en daar de actie afwerken.

- Een **'dictionary'** om identifiers en hun waarde in op te slaan. De waarden zijn ofwel 'inline' of referenties naar een deelverzameling van mogelijke expressies (waarden zijn eigenlijk expressies die een identiteit hebben voor de evaluatie functie). Een Borg 'dictionary' is een stack met frames om zo de notie van scope te kunnen ondersteunen.
- Een **evaluatie-loop** die de evaluatie functies tot de borg semantiek verzameld. Deze loop aanvaardt een sequentie van expressies en evalueert deze.
- Een **router** die zorgt dat agents berichten naar elkaar kunnen sturen en toelaat dat agents naar andere hosts kunnen migreren.

Het volledige proces zal programma tekst aanvaarden en deze converteren naar een expressie die doorgegeven wordt aan een gespecialiseerde interpreter. Bijv.:

```
if(exp.operand='-',
 {
  par : evaluate(exp.par(1));
  number(-par.value)
 },
 if(exp.operand='+',
  {
   par1 : evaluate(exp.par(1));
   par2 : evaluate(exp.par(2));
   number(par1.value+par2.value)
  },
  error()))
```

dan, bijvoorbeeld, de evaluatie van

```
apply(-,apply(+,1,2))
```

zal resulteren in een oproep sequentie van

```
apply(apply(-,apply(+,1,2)))
evaluate(apply(+,1,2))
evaluate(1)
evaluate(2)
```

De expressies die gebruikt werden in deze uitwerking worden meestal bewaard op een stack, dewelke meestal samenvalt met de run-time stack van het programma die de interpreter implementeert.

3.4 Borg Computational Model

In Borg zien we het computationeel-model als een expressie/continuatie stack. Hieronder zijn de regels beschreven van de bestaande interpreter :

- Om de parameters van een functie te krijgen, moet het deze zelf gaan afhalen van de expressie stack (met peek & pop[2]).
- Als andere functies moeten opgeroepen worden (zoals evaluate bijvoorbeeld), dan zal het deze functies (en bijhorende expressie) op de continuatie stack bewaren.
- Op het einde van een gegeven functie, moet het verder gaan met de volgende functie die op de continuatie stack te vinden is. In een staart-recursieve taal kan dit gedaan worden door contstack.pop() op te roepen op het einde van de functie. Als we geen staart-recursieve taal hebben, moeten we de functie beeindigen (return;) en hopen dat er een 'outer-stack-loop' voor handen is.

De geinterpreteerde code van -(1+2) zal geconverteerd worden tot :

```
Minus()::
 {
  par: expstack.pop();
  expstack.push(number(-par.value))
 }
Apply()::
 {
  exp: expstack.pop();
  if(exp.operand = '-',
  {
   constack.push(minus);
```

[2]peek() : bovenste element op de stack bekijken maar op de stack laten staan.
pop() : bovenste element van de stack afhalen.

```
        constack.push(evaluate);
        expstack.push(exp.par(1))
      },
      if (exp.operand = '+',
      {
        par1 : exp.par(1);
        par2: exp.par(2);
        constack.push(addfinal);
        constack.push(addaux);
        expstack.push(par2);
        constack.push(evaluate);
        expstack.push(par1)
      },
      error()))
    }
  Addaux()::
    {
      result1 : expstack.pop();
      exp2: expstack.pop();
      expstack.push(result1);
      expstack.push(exp2);
      contstack.push(evaluate)
    }
  Addfinal()::
    {
      result1 : expstack.pop();
      result2: expstack.pop();
      expstack.push(number(result1.value+result2.value))
    }
```

de gebruikte stacks in een soort van root-tabel steken.

- We hebben een extra 'indirection' welke kan leiden tot een 'suboptimal' performantie.
- We kunnen gemakkelijk stack optimizaties implementeren, want we hebben de stack onder controle. Bijvoorbeeld staart-recursie implementeren is geen probleem want we veranderen gewoon de 'apply' een beetje : een applicatie bestaat uit het veranderen van de huidige dictionary, het evalueren van een expressie en het terug geven van het resultaat. Als we controlleren dat er reeds een 'return-result' continuatie is, dan hoeven we geen nieuwe meer te pushen.

3.5 Borg intern

Borg bestaat uit 2 grote delen. Het eerste deel is de borgcore, het systeem dat de agenten interpreteert (de virtuele machine). Het tweede deel zijn de verschillende gebruikersinterfaces voor de verschillende platvormen (Windows, Linux, Apple, Palm, tekst interface, web interface, ...).

3.5.1 Algemeen

Borg werd geschreven in ANSI C. Dit omdat ANSI C een zeer 'portable' taal is en daarmee ook platvorm onafhankelijk is. We kunnen borg dus uitvoeren op meerdere platvormen waar er een Userinterface voor handen is. Zo kunnen onze agenten die in Borg geschreven zijn ook op meerdere platvormen rondwandelen.

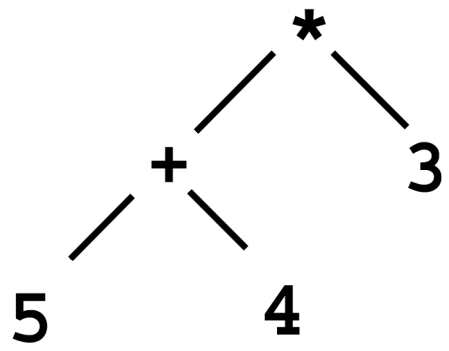
3.5.2 Continuaties

Borg is gebaseerd op een 'pushdown automaat' die expressies en continuaties/thunks managed op 2 stacks. Een continuatie is een stuk op zichzelf staande code dat een actie uitvoert, zonder een return-value!

Voorbeeld :

```
continuatiename()
  {
    get_value_from_stack(a);
    put_value_on_stack(a+1);
  }
```

Deze continuaties worden gemanaged op een stack. Borg heeft 2 stacks. Een van de stacks houdt de continuaties bij, de andere houdt de data bij. Een



Figuur 6: boom van (5 + 4) * 3

```
/*-------------------------------------------------------------------*/
/*  MUL                                                              */
/*     EXP: [... ... ... ... ... ARG] -> [... ... ... ... EXP EXP]  */
/*     CNT: [... ... ... ... ... MUL] -> [... ... ... MU1 SWP EXP]  */
/*-------------------------------------------------------------------*/
_NIL_TYPE_ MUL(_NIL_TYPE_)
  { BINARY(__MU1, MUL_str); }
```

Figuur 7: MUL continuatie uit NatArithmetic.c

stack werkt volgens het FILO (First In, Last Out) principe, en dus moeten we de laatste actie als eerst op de stack zetten. Een voorbeeld zal duidelijk maken hoe deze stacks en continuaties werken.
We willen bijvoorbeeld de expressie (5 + 4) * 3 uitwerken.
De parser construeert een boom zoals in figuur 6.
Aan de hand van deze figuur zien we dat we eerst de vermenigvuldiging moeten uitwerken(bovenste node). **Opmerking :** We hebben een paar stappen overgeslagen voor de simpliciteit en starten direct bij de uitwerking van de vermenigvuldiging.

We zoeken de continuatie van de vermenigvuldiging op. figuur 7.

Deze continuatie vertelt ons dat we eerst 'BINARY' moeten oproepen. figuur 8.

Binary is een macro die ons vertelt dat we 'BINARY_BODY' moeten oproepen. figuur 9.

```
#define BINARY(CNT, NAM) BINARY_BODY(CNT,NAM);
```

Figuur 8: BINARY macro uit Natives.h

```
#define BINARY_BODY(CNT, NAM)\
  _EXP_TYPE_ arg, exp;\
  _UNS_TYPE_ siz;\
  _stk_claim_();\
  _stk_peek_EXP_(arg);\
  siz = _ag_get_TAB_SIZ_(arg);\
  if (siz == 2)\
    { exp = _ag_get_TAB_EXP_(arg, 2);\
      _stk_poke_EXP_(exp);\
      exp = _ag_get_TAB_EXP_(arg, 1);\
      _stk_push_EXP_(exp);\
      _stk_poke_CNT_(CNT);\
      _stk_push_CNT_(__SWP);\
      _stk_push_CNT_(__EXP);\
      return; }\
  _error_str_(_NMA_ERROR_, NAM)
```

Figuur 9: BINARY_BODY macro uit Natives.h

Als parameters van BINARY_BODY kregen we 'MU1' mee en 'MUL_str'. De eerste parameter, 'MU1', is de naam van de volgende continuatie, 'MUL_str' is voor de foutmelding, en heeft weinig belang voor deze uitleg. BINARY_BODY haalt de argumenten van de vermenigvuldiging van de data stack (_stk_peek_EXP_(arg))[3] en controleert of er wel degelijk 2 argumenten zijn. Deze 2 argumenten zitten in een tabel. Als er 2 argumenten gevonden worden, worden deze uit de tabel gehaald en terug op de data-stack gezet, waarbij de tabel overschreven wordt, zodat deze verdwijnt. Op de continuatie-stack wordt eerst de continuatie die laatst moet uitgevoerd worden, gezet (en overschrijft de vorige top van de stack). Dit is de 'MU1'-continuatie. Vervolgens wordt de 'SWP' (SWAP) continuatie op de stack gezet, en bovenaan komt 'EXP' continuatie. Dan is 'BINARY_BODY' uitgewerkt. Dus wordt de volgende top van de stack gehaald en uitgevoerd. Dit is de 'EXP' continuatie. figuur 10.

Deze 'EXP' continuatie haalt eerst van de data-stack het bovenste argument eraf. Dit is het eerste argument van onze bewerking. En hij kijkt wat voor soort expressie dit is. Daar het om een optelling gaat, is dit een '_APL_TAG_' en poked hij op de continuation stack : 'APL'. Nu is ook de 'EXP'-continuatie uitgewerkt. En staat de 'APL'-continuatie bovenaan. figuur 11

APL haalt van de expressie stack de bovenste expressie af en we weten dat

[3]peek: Enkel kijken wat er op de stack staat.
pop: Bovenste element van de stack afhalen.
poke: Bovenste element vervangen door nieuw element.
push: Bovenop het bovenste element het nieuwe element zetten.
zap: Het bovenste element van de stack weggooien.

```
/*-------------------------------------------------------------------*/
/*  EXP                                                              */
/*     EXP: [... ... ... ... ... EXP] -> [... ... ... ... ... EXP]  */
/*     CNT: [... ... ... ... ... EXP] -> [... ... ... ... ... ***]  */
/*                                                                   */
/*     EXP: [... ... ... ... ... EXP] -> [... ... ... ... ... EXP]  */
/*     CNT: [... ... ... ... ... EXP] -> [... ... ... ... ... ...]  */
/*-------------------------------------------------------------------*/
_NIL_TYPE_ EXP(_NIL_TYPE_)
{
   _EXP_TYPE_ exp;
   _stk_peek_EXP_(exp);
   switch (_ag_get_TAG_(exp))
     {
      case _APL_TAG_: _stk_poke_CNT_(__APL); return;
      case _DCL_TAG_: _stk_poke_CNT_(__DCL); return;
      case _DEF_TAG_: _stk_poke_CNT_(__DEF); return;
      case _MES_TAG_: _stk_poke_CNT_(__MES); return;
      case _REF_TAG_: _stk_poke_CNT_(__REF); return;
      case _SET_TAG_: _stk_poke_CNT_(__SET); return;
      case _TBL_TAG_: _stk_poke_CNT_(__TBL); return;
     }
   _stk_zap__CNT_();
}
```

Figuur 10: EXP continuatie uit Evaluator.c

deze van het type APL is. Een APL heeft 3 delen, nl. een naam, argumenten en een dictionary. In deze dictionary kunnen we opzoeken welke functie hier mee overeen stemt. Deze functie wordt op de expressie stack gepoked en de argumenten worden er bovenop gepushed. Op de CNT-stack wordt ofwel de NAT (native) continuatie gepoked ofwel de CLL (functie) continuatie. Daar het hier om een plus gaat, moeten wij de NAT-continuatie op de stack poken. figuur 12.

Dus nu moeten we de NAT-continuatie uitwerken. In deze NAT-continuatie wordt eerst de argumenten van de EXP-stack gehaald en dan de native-functie. De argumenten worden direct terug op de EXP-stack gepoked. Er wordt gekeken welke native functie we nu eigenlijk hebben, en de overeenkomstige continuatie wordt op de CNT-stack gepoked. Daar we een '+' hebben, wordt de ADD-thunk op de CNT-stack gepoked. figuur 13

Uitwerking van ADD. ADD vertelt ons dat we UNABINARY moeten oproepen. figuur 14.

UNABINARY is een macro waarbij nagegaan wordt of de plus-operator 1 argument heeft (+5 is een geldige wiskundige expressie) of 2 argumenten. In ons geval zijn het er 2 en zullen deze 2 argumenten uit hun tabel gehaald worden en op de EXP-stack gestoken worden. Op de CNT-stack wordt AD2,

```
/*-------------------------------------------------------------------*/
/*  APL                                                              */
/*     EXP: [... ... ... ... ... APL] -> [... ... ... ... FUN ARG]  */
/*     CNT: [... ... ... ... ... APL] -> [... ... ... ... ... CLL]  */
/*                                                                   */
/*     EXP: [... ... ... ... ... APL] -> [... ... ... ... FUN ARG]  */
/*     CNT: [... ... ... ... ... APL] -> [... ... ... ... CLL EXP]  */
/*                                                                   */
/*     EXP: [... ... ... ... ... APL] -> [... ... ... ... NAT ARG]  */
/*     CNT: [... ... ... ... ... APL] -> [... ... ... ... ... NAT]  */
/*                                                                   */
/*     EXP: [... ... ... ... ... APL] -> [... ... ... ... NAT ARG]  */
/*     CNT: [... ... ... ... ... APL] -> [... ... ... ... NAT EXP]  */
/*-------------------------------------------------------------------*/

_NIL_TYPE_ APL(_NIL_TYPE_)
{
   _EXP_TYPE_ apl, arg, nam, fun, dct;
   _stk_claim_();
   _stk_peek_EXP_(apl);
   nam = _ag_get_APL_NAM_(apl);
   arg = _ag_get_APL_ARG_(apl);
   dct = _dct_locate_(nam, _DCT_);
   fun = _ag_get_DCT_VAL_(dct);
   _stk_poke_EXP_(fun);
   _stk_push_EXP_(arg);
   switch (_ag_get_TAG_(fun))
     { case _FUN_TAG_: _stk_poke_CNT_(__CLL); break;
      case _NAT_TAG_: _stk_poke_CNT_(__NAT); break;
      default: _error_msg_(_NAF_ERROR_, nam); }
   if (!_ag_is_TAB_(arg)) _stk_push_CNT_(__EXP);
}
```

Figuur 11: APL continuatie uit Evaluator.c

```
/*-------------------------------------------------------------------*/
/*  NAT                                                              */
/*     EXP: [... ... ... ... NAT TAB] -> [... ... ... ... ... TAB]  */
/*     CNT: [... ... ... ... ... NAT] -> [... ... ... ... ... ***]  */
/*-------------------------------------------------------------------*/
_NIL_TYPE_ NAT(_NIL_TYPE_)
{  _EXP_TYPE_ thk, nat, tab;
   _stk_pop__EXP_(tab);
   _stk_peek_EXP_(nat);
   _stk_poke_EXP_(tab);
   _trace_nat_(nat,tab);
   thk = _ag_get_NAT_THK_(nat);
   if (!_ag_is_TAB_(tab))
     _error_str_(_IAG_ERROR_, _ag_get_NAT_NAM_(nat));
   _stk_poke_CNT_(thk); }
```

Figuur 12: NAT continuatie uit Evaluator.c

```
/*-------------------------------------------------------------------*/
/*  ADD                                                              */
/*     EXP: [... ... ... ... ... ARG] -> [... ... ... ... EXP EXP]  */
/*     CNT: [... ... ... ... ... ADD] -> [... ... ... AD2 SWP EXP]  */
/*                                                                   */
/*     EXP: [... ... ... ... ... ARG] -> [... ... ... ... ... EXP]  */
/*     CNT: [... ... ... ... ... ADD] -> [... ... ... ... AD1 EXP]  */
/*-------------------------------------------------------------------*/
_NIL_TYPE_ ADD(_NIL_TYPE_)
{
   UNABINARY(__AD1, __AD2, ADD_str);
}
```

Figuur 13: ADD continuatie uit NatArithmetic.c

```
#define UNABINARY(CNU, CNB, NAM)\
  _EXP_TYPE_ arg, exp;\
  _UNS_TYPE_ siz;\
  _stk_claim_();\
  _stk_peek_EXP_(arg);\
  siz = _ag_get_TAB_SIZ_(arg);\
  switch (siz)\
    { case 1:\
        exp = _ag_get_TAB_EXP_(arg, 1);\
        _stk_poke_EXP_(exp);\
        _stk_poke_CNT_(CNU);\
        _stk_push_CNT_(__EXP);\
        return;\
      case 2:\
        exp = _ag_get_TAB_EXP_(arg, 2);\
        _stk_poke_EXP_(exp);\
        exp = _ag_get_TAB_EXP_(arg, 1);\
        _stk_push_EXP_(exp);\
        _stk_poke_CNT_(CNB);\
        _stk_push_CNT_(__SWP);\
        _stk_push_CNT_(__EXP);\
        return; }\
  _error_str_(_NMA_ERROR_, NAM);
```

Figuur 14: UNABINARY macro uit Natives.h

SWP en EXP geplaatst. Deze laatste continuatie is reeds uitgelegd en heeft tot doel om te kijken welke expressie op de stack te vinden is (in dit geval een getal). De SWP zal de 2 bovenste argumenten van de EXP-stack van plaats verwisselen, en een EXP-continuatie op de CNT-stack poken. Nu zal het 2de argument van de optelling uitgewerkt worden, en daar dit een getal is, gebeurt er niets mee. We vinden nu op de EXP-stack 2 getallen.

AD2 zal de optelling uitwerken door na te gaan welk type de 2 getallen zijn, en deze dan bij elkaar op te tellen. Dit resultaat wordt op de EXP-stack gepoked. figuur 15.

Nu staat bovenaan de SWP-continuatie. Dit om het 2de argument van de vermenigvuldiging te evalueren. Deze zal de 2 bovenste argumenten van de EXP-stack van plaats verwisselen, en een EXP-continuatie op de CNT-stack poken. Nu zal het 2de argument van de vermenigvuldiging uitgewerkt worden, en daar dit een getal is, gebeurt er niets mee. figuur 16.

MU1 vinden we nu bovenaan, en MU1 zal de vermenigvuldiging uitwerken, rekening houdend met de types van de getallen. Dit resultaat wordt bovenaan op de EXP-stack geplaatst, en dit is dan ook de plaats waar we de uitkomst vinden. figuur 17.

3.6 Strong Mobility

Na het design van dichterbij bekeken te hebben, gaan we eens dieper in op strong mobility (sterke mobiliteit). Want Borg werd gecreeerd om sterke mobiliteit te ondersteunen. Borg is een sterk mobiel platvorm. Dit wil zeggen dat we Borg-agenten kunnen verplaatsen op ons platvorm naar een andere locatie en dit op elk moment tijdens hun uitvoering, zelfs binnen lussen en geneste structuren. Dit doel wordt bereikt door middel van volgende 5 stappen :

1. Eerst moet de computationele-staat van de agent gevat worden.
2. Dan moet deze geserialiseerd worden,
3. en over het netwerk gestuurd worden naar de nieuwe bestemming.
4. Uiteindelijk moet het gedeserialiseerd worden,
5. en de uitwerking moet/kan verder gaan.

Alhoewel dit algoritme zeer gemakkelijk lijkt, is het toch moeilijk in praktijk te brengen. Vooral het vatten van deze computationele staat blijkt problematisch. Deze computationele staat bestaat meestal uit een soort stack,

```
/*-------------------------------------------------------------------*/
/*  AD2                                                              */
/*     EXP: [... ... ... ... VAL VAL] -> [... ... ... ... ... VAL] */
/*     CNT: [... ... ... ... ... AD2] -> [... ... ... ... ... ...] */
/*-------------------------------------------------------------------*/
_NIL_TYPE_ AD2(_NIL_TYPE_)
{
   _EXP_TYPE_ frc, nbr, txt, va1, va2;
   _UNS_TYPE_ len;
   _FLP_TYPE_ fl1, fl2;
   _STR_TYPE_ st1, st2;
   _SGN_TYPE_ sg1, sg2, sgn;
   _FLO_TYPE_ flo;
   _mem_claim_();
   _stk_pop__EXP_(va2);
   _stk_peek_EXP_(va1);
   _stk_zap__CNT_();
   if (_ag_is_NBR_(va1))
     {
        sg1 = _ag_get_NBR_(va1);
        if (_ag_is_NBR_(va2))
          {
             sg2 = _ag_get_NBR_(va2);
             sgn = sg1 + sg2;
             if (labs(sgn) <= _NBR_MAX_)
               {
                  nbr = _ag_make_NBR_(sgn);
                  _stk_poke_EXP_(nbr);
                  return;
               }
             /* JF - Here - to test */
             _stk_push_EXP_(va2);
             _stk_push_CNT_(__AD2);
             _error_str_(_NBR_ERROR_, ADD_str);
             return;
          }
   <knip>
```

Figuur 15: AD2 continuatie uit NatArithmetic.c

```
/*-------------------------------------------------------------------------*/
/*  SWP                                                                    */
/*   exp-stk: [... ... ... ... EXP VAL] -> [... ... ... ... VAL EXP]  */
/*   cnt-stk: [... ... ... ... ... SWP] -> [... ... ... ... ... EXP]  */
/*-------------------------------------------------------------------------*/
_NIL_TYPE_ SWP(_NIL_TYPE_)
{
   _EXP_TYPE_ val, exp;
   _stk_pop__EXP_(val);
   _stk_peek_EXP_(exp);
   _stk_poke_EXP_(val);
   _stk_push_EXP_(exp);
   _stk_poke_CNT_(__EXP);
}
```

Figuur 16: SWP continuatie uit Natives.c

geheugen en de code. Voor een gecompileerde taal lijkt de computationele staat meer op een data stack, een handvol registers en de code. Dus als we de 'computational state' willen vatten, moeten we een kopie nemen van de interne stack van de virtuele machine, het geheugen-model en de code. De code vatten is meestal niet zo een groot probleem omdat deze reeds in de meeste talen toegankelijk werd gemaakt via een abstracte grammatica. Maar het vatten van de stack en het geheugen is problematischer en verplicht meestal de programmeur ertoe zelf een kopie van de interne data van de virtuele machine bij te houden. In de praktijk zien we dat sterke mobiliteit geimplementeerd is in de meeste programmeertalen, hoewel de schoonheid en de mogelijke tekortkomingen meestal sterk verbonden zijn met de reflectieve natuur van de programmeertaal.

Een ander probleem bij strong mobility is de communicatie tussen agenten werkende te houden. Deze agenten kunnen zich vrij op het netwerk begeven, maar toch moeten boodschappen hen kunnen bereiken.

3.7 Conclusie

In dit hoofdstuk zijn we dieper ingegaan op Pico/Borg.

Borg is een continuatie gebaseerde programmeertaal die ontwikkeld werd aan de Vrije Universiteit Brussel. Het is een mobiel multi-agent platvorm waarbij het geheel gebaseerd is op continuaties. We hebben aan de hand van een voorbeeld de werkwijze van Borg gedemonstreert.

Borg is daarenboven ook nog een sterk mobiel platvorm, wat wil zeggen dat agenten midden in een berekening, deze kunnen stoppen, zich verplaatsen naar een andere machine en daar de uitwerking verder zetten.

```
/*-------------------------------------------------------------------*/
/*  MU1                                                              */
/*     EXP: [... ... ... ... VAL VAL] -> [... ... ... ... ... VAL] */
/*     CNT: [... ... ... ... ... MU1] -> [... ... ... ... ... ...] */
/*-------------------------------------------------------------------*/
_NIL_TYPE_ MU1(_NIL_TYPE_)
  { _EXP_TYPE_ frc, nbr, va1, va2;
    _FLP_TYPE_ fl1, fl2;
    _SGN_TYPE_ sg1, sg2, sgn;
    _FLO_TYPE_ flo;
    _mem_claim_();
    _stk_pop__EXP_(va2);
    _stk_peek_EXP_(va1);
    _stk_zap__CNT_();
    if (_ag_is_NBR_(va1))
      { sg1 = _ag_get_NBR_(va1);
        if (_ag_is_NBR_(va2))
          { sg2 = _ag_get_NBR_(va2);
            flo = (_FLO_TYPE_)sg1 * (_FLO_TYPE_)sg2;
            if (fabs(flo) > _NBR_MAX_)
              {
                /* JF - Here - to test */
                _stk_push_EXP_(va2);
                _stk_push_CNT_(__MU1);
                _error_str_(_NBR_ERROR_, MUL_str);
                return; }
            sgn = (_SGN_TYPE_)flo;
            nbr = _ag_make_NBR_(sgn);
            _stk_poke_EXP_(nbr);
            return; }
        if (_ag_is_FRC__(va2))
          { fl2 = _ag_get_FRC_(va2);
            flo = sg1 * *fl2;
            frc = _ag_make_FRC_(&flo);
            _stk_poke_EXP_(frc);
            return; }}
    else if (_ag_is_FRC__(va1))
      { fl1 = _ag_get_FRC_(va1);
        if (_ag_is_NBR_(va2))
          { sg2 = _ag_get_NBR_(va2);
            flo = *fl1 * sg2;
            frc = _ag_make_FRC_(&flo);
            _stk_poke_EXP_(frc);
            return; }
        if (_ag_is_FRC__(va2))
          { fl2 = _ag_get_FRC_(va2);
            flo = *fl1 * *fl2;
            frc = _ag_make_FRC_(&flo);
            _stk_poke_EXP_(frc);
            return; }}
    /* JF - Here - to test */
    _stk_push_EXP_(va2);
    _stk_push_CNT_(__MU1);
    _error_atc2_(MUL_str,va1,va2); }
```

Figuur 17: MU1 continuatie uit NatArithmetic.c

Dit maakt van Borg een goed startpunt om deze thesis op te bouwen.

4 Jit-compilatie

> Those who cannot remember the past are condemned to repeat it. – **George Santayana, 1863-1952**

Het is de bedoeling om een reflectieve virtuele machine te maken door middel van JIT-compilatie, dan is het ook aangewezen om eens wat dieper in te gaan op Jit-compilatie. Aan de hand van de Java jit-compiler zullen we de voordelen van jit-compilatie uit de doeken doen.

4.1 inleiding

Strikt gesproken zijn JIT-compilatie systemen (JIT in het kort) volledig nutteloos. Ze zijn enkel nodig om de performantie van programma's te verbeteren en de plaats te verkleinen die deze programma's innemen op een systeem. Tenslotte is het probleem dat JIT oplost eigenlijk al een opgelost probleem : het vertalen van programma's van een programmeertaal naar een vorm die uitvoerbaar is op het doel-platvorm.

Traditioneel zijn er 2 manieren om bron-code te vertalen : compilatie en interpretatie. Compilatie vertaalt van een brontaal naar een doeltaal - Bijv. C naar assembler-code, met het doel dat de vertaalde vorm handelbaarder is voor latere uitvoering. Interpretatie elimeneert deze tussenstappen, maar voert dezelfde analyse uit en voert de handeling onmiddellijk uit, dit met het nadeel dat het trager is.

Jit compilatie wordt gebruikt om de voordelen van deze 2 benaderingen (compilatie & interpretatie) te benutten. Een kleine samenvatting van deze voordelen :

- Gecompileerde programma's worden veel sneller uitgevoerd, vooral als zij gecompileerd zijn naar een vorm die direct uitvoerbaar is op de onderliggende hardware. Statische compilatie kan ook een optimizatie uitvoeren door het programma te analyseren. Dit brengt ons tot een eerste doel van JIT systemen : snelheid. Een Jit systeem mag geen ongunstige pauses veroorzaken in een normale programma-uitvoering door zijn operaties.

- Geinterpreteerde programma's zijn typisch veel kleiner, doordat een representatie gekozen is die op een hoger niveau staat dan machine-code, en zo meer semantiek kan hebben in zijn vorm.

- Geinterpreteerde programma's zijn meestal meer 'portable' naar andere systemen. Dit doordat een machine-onafhankelijke representatie van de

broncode werd genomen. Enkel de interpreter moet op de verschillende platvormen kunnen werken.

- Interpreters hebben toegang tot 'at-runtime' informatie, zoals de input parameters, die ongeweten zijn bij statische programma analyse.

We moeten hierbij wel afwegen welke optimisaties we gaan doorvoeren bij het compileren, want deze optimisaties kunnen de uitvoeringstijd negatief beinvloeden.

4.2 Java & JIT-compilatie

Java is gekend als een geinterpreteerde taal. Dit is maar voor de helft waar, want de java-bron code wordt gecompileerd in een intermediate code, beter bekend als Java bytecode of J-code. Het verschil met andere compilers is dat andere compilers compileren naar een intermediate code die verstaanbaar is voor de compiler en die later gelinked wordt tot libraries of executables. Java bytecode daarentegen wordt later nog geinterpreteerd door de Java Virtual Machine.

De reden dat Java-code wordt omgezet in bytecode is dat deze bytecode kan geinterpreteerd worden door elke virtuele machine die goed geconstrueerd werd.

Interpretatie is veel trager dan compilatie, en dus zijn java-programma's traag. Om aan dit euvel te verhelpen, worden JIT-compilers gebruikt. Jit-compilers combineren de voordelen van interpretatie en compilatie. We doen dit door 'juist-op-tijd' onze code te compileren. "Juist-op-tijd" wil zeggen : tijdens de uitvoering en dat op het moment dat de code wordt opgeroepen. We kunnen dus de voordelen van de interpretatie aanwenden en bijv. at runtime optimizaties uitvoeren. Dit wordt nu gecombineerd met de voordelen van compilatie en dat wil dus zeggen dat, eenmaal dezelfde code verder in het programma opnieuw wordt opgeroepen, deze niet meer geinterpreteerd/gecompileerd hoeft te worden. De jit-compiler weet dat hij deze code al eens gecompileerd heeft, en zal deze reeds gecompileerde code gebruiken.

4.3 Java Jit compiler overzicht

Om Just-in-time compilers beter te verstaan, zullen we eerst kijken hoe de Java Virtuele machine werkt. Java is nl. de taal die Jit-compilatie terug op de voorgrond gebracht heeft. De technologie van Jit-compilatie bestaat al heel wat langer dan vandaag, maar omdat java-programma's eerst gecompileerd worden naar bytecode en dat deze bytecode later geinterpreteerd dient te

worden op een Java virtuele machine, heeft men Jit opnieuw opgerakeld. Jit zal de interpretatie van de bytecode vervangen, en zal een performantie winst geven.

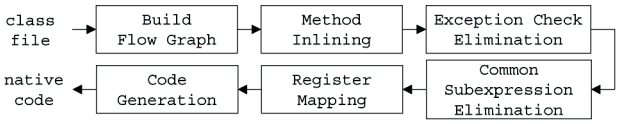
We zullen even de stappen volgen die een java programma doorloopt. Als men een Java programma schrijft zoals het volgende "hello world" programma:

```
class hello
 {
  public static void main(String argv[])
  {
   System.out.println("Hello World!");
  }
 }
```

dan runt men eerst "javac", de Java compiler. Deze vormt de code om in wat gekend is als bytecode en steekt deze in een "hello.class" file. Deze class file kan dan geinterpreteerd worden op elke machine die een Java virtuele machine heeft. Het keyword hier is interpretatie. De Java Virtuele machine processed elke bytecode in de .class file en voert deze uit. Dit is hetzelfde zoals andere geinterpreteerde talen doen zoals SCHEME en SMALLTALK.

Wanneer er een JIT aanwezig is zal de virtuele machine iets anders doen. Nadat deze de .class file heeft ingelezen voor interpretatie, zal het deze .class file doorgeven aan de JIT. De JIT zal de bytecodes nemen en deze compileren naar native-code voor de machine waarop men aan het werken is. Het kan sneller zijn om deze code te compileren en ze uit te voeren dan gewone interpretatie omdat men de terugkomende code reeds gecompileerd heeft en niet opnieuw moet interpreteren. Ook zullen enkel de nodige stukken code gecompileerd worden en alles wat niet nodig is wordt niet gecompileerd. De JIT is een deel van de Java Virtuele machine, en men zal dus niet weten dat er een JIT aanwezig is, behalve dat de code sneller zal werken. Sommige omgevingen laten toe om te kiezen voor JIT-compilatie of niet.

Java is een dynamische taal, dus is het niet toegelaten om statisch te compileren. Dit wil zeggen dat het niet toegelaten is alle .class files om te zetten naar machine code. Dit gebeurt enkel als de code opgeroepen wordt. Dus de JIT is echt wel 'juist op tijd' omdat het de methodes compileert juist voor dat deze opgeroepen worden. Als men meer dan een keer deze methode gebruikt zal de JIT niet opnieuw compileren, maar de reeds gecompileerde code gebruiken. Dit zal dus heel wat sneller gaan.



Figuur 18: Een overzicht van de Jit-compiler

Jit zal meestal een snelheidswinst opleveren. In sommige gevallen kan het eens voorkomen dat interpretatie sneller zou zijn. Hier komt dan de Smart-Jit in het verhaal. De Smart-Jit zal kijken of interpretatie inderdaad niet sneller zou zijn. Methodes die maar 1 keer gebruikt worden zouden inderdaad meer tijd in beslag nemen als men ze compileert. Daarvoor is interpretatie sneller. De werkwijze van de Smart-Jit bestaat eruit om te tellen hoeveel keer een methode opgeroepen wordt, en indien een aantal overschreden wordt, zal de Smart-Jit overgaan tot compilatie.

De Java JIT-compiler vertaalt de bytecode naar native-code in 6 fasen. (figuur 18) Eerst construeert het de basis blokken en de loop-structuur van de bytecode. Dan voert het inlining uit op zowel de statische als dynamische methode oproepen. Inlining van dynamische methode-oproepen wordt uitgevoerd gebruik makende van een klasse hierarchie analyse. De JIT compiler verwijdert dan exception checks en voert ook nog andere optimizaties uit zoals constante propagatie en dode code eliminatie. Na dit verwijdert de JIT de gemeenschappelijke subexpressies om zo het aantal toegangen tot arrays en instance variabelen te verminderen. We zien hier dat we de bytecode uitbreiden om zo de interne pointer van het object te representeren en zo de consecutieve array toegang verbeteren. Vervolgens zal de JIT compiler elke stack operand mappen aan ofwel een logische integer of een floating point register, en telt het aantal gebruiken van lokale variabelen in elke regio van een programma. Deze regio's worden bepaald op basis van de loop structuur in deze fase. Uiteindelijk genereert het de native-code van de byte code tesamen met een fysische register allocator. Daar de JIT compiler nood heeft aan snelle compilatie kunnen dure register-allocation algoritmen, zoals graph coloring, niet gebruikt worden. In plaats daarvan gebruikt het een snel algoritme om registers toe te wijzen zonder een extra fase in te voegen. In elke region worden vaak gebruikte locale variabelen toegewezen aan fysische registers. De overige registers worden gebruikt voor stack operands die gebruikt worden tijdens de computation. Indien de codegenerator dan nood heeft aan een nieuw register, maar er geen registers meer voor handen zijn, dan zal de register allocator het minst gebruikte register vinden en deze daarvoor gebruiken. Zo worden dure zoekmethoden voor de 'spill' kandidaten vermeden. Live informatie van lokale variabelen, verkregen uit de data flow analyse

wordt ook gebruikt om generatie van inefficiente code te vermijden.

Een ander voordeel van Jit-compilatie is dat het gehele programma zich nog niet in de virtuele machine moet bevinden (bijvoorbeeld in gedistribueerde systemen) maar het programma wel al opgestart kan worden. De jit-compiler compileert enkel maar als het nodig is, en als deze code nog niet gedownloaded zou zijn, dan wacht de jit-compiler tot de code binnen is.

4.4 Conclusie

In dit hoofdstuk hebben we kennis gemaakt met Jit-compilatie. Jit compilatie is een handige techniek om interpretatie gebaseerde talen te versnellen door middel van compilatie tijdens de uitvoering zodat dubbel gebruikte code reeds in een performante vorm te vinden is. Het combineert de voordelen van compilatie, nl. een snelle uitvoering, en interpretatie, nl. at runtime optimisatie mogelijkheden & kleinere broncode.

Deel II
Huidige RVM's

5 RbCl

> I'm willing to admit that I may not always be right, but I'm never wrong! – **anonymous**

Tot nu toe is Borg een systeem waarbij het de bedoeling is om features van Borg at runtime te kunnen veranderen. Deze taal heeft initieel een semantiek en syntax. We zullen dit de kernel noemen. Als we onze virtuele machine opstarten wordt de kernel met zijn eigenschappen ingeladen en kunnen we starten. RbCl (a Reflective based Concurrent language) is een systeem zonder een run-time kernel. Dit wil zeggen dat alles, behalve hetgeen het OS aangaat, zich in een metacirculaire vorm bevindt en wordt ingeladen. Dit heeft als gevolg dat ook alles, behalve hetgeen het OS aangaat, kan aangepast worden. Dit reflectief systeem kan de data, Causally-Connected Self Representation (CCSR) genaamd, manipuleren. De CCSR stelt de huidige staat van zijn computation voor.

Vorige reflectieve systemen hebben volgende problemen :

- Alle vorige reflectieve talen werden gebouwd op hun run-time kernels. Deze kunnen niet aangepast worden door de gebruiker. Hoewel de semantiek misschien aanpasbaar is door middel van reflectie, kan het gedrag van de runtime-kernel niet aangepast worden. Dit is een probleem voor de taal-gebruikers die verzot zijn op efficientie: bijvoorbeeld zou een gebruiker de taal willen aanpassen aan een specifiek probleem om zo de efficientie te verbeteren. Toch zal het bestaan van de runtime kernel zijn veranderingen minimaliseren. Ook kan de reflectiviteit van de taal niet veranderd worden, omdat deze ingebakken zit in de runtime kernel.

- In vele talen waarin de CCSR kan aangepast worden, zal niet het volledige aspect van de implementatie blootgeven worden. De CCSR is meestal geabstraheerd dat de gebruiker enkel het gedrag van het systeem kan veranderen. Dit beperkt de aspecten van de uitwerking die door gebruiker kunnen veranderd worden. Bijvoorbeeld zal het mechanisme van de garbage-collection in de runtime kernel zitten, en dus niet aanpasbaar zijn.

- In de meeste reflectieve systemen zal de reflectieve-toren die door de gebruiker gezien wordt, niet bestaan. De runtime kernel zal eerder deze toren van interpreters simuleren. Omdat de reflectieve-toren ver van de actuele implementatie staat, zal de gebruiker het gedrag van de run-time kernel moeten kennen om te kunnen voorspellen hoeveel CPU-tijd en geheugen er nodig is voor een reflectief programma. Daarenboven zullen complexe implementaties nodig zijn voor efficiente implementaties hoewel hun metacirculaire voorstelling heel wat simpeler zou zijn.

RbCl is een taal die deze nadelen wil verhelpen. Deze taal voldoet aan de volgende eigenschappen :

- Een simpel mechanisme, System Object Table genaamd, zal de huidige runtime kernel vervangen op vlak van de implementatie taal. In andere woorden : het zal mogelijk zijn het volledige gedrag van het systeem te vervangen buiten de routines die betrekking hebben tot het OS.

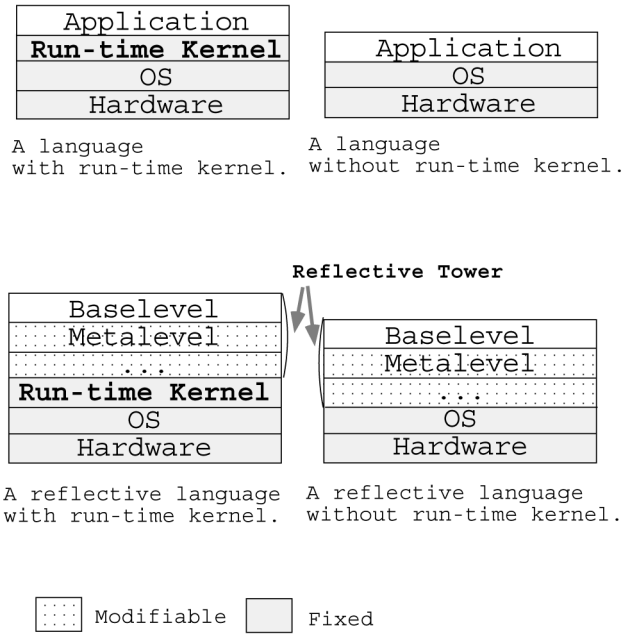
- Een nieuwe vaardigheid wordt geintroduceerd, nl. linguistic symbiosis met de implementatie taal om zo de CCSR volledig in overeenstemming te brengen met de huidige implementatie.

- in RbCl kunnen we de reflectieve-toren wel bezien als echt bestaand. De reflectieve-toren van RbCl is de oneindige toren van de directe implementatie. Het metasysteem wordt echter gecreeerd op een lazy manier. Deze creatie van metasystemen kan gemaakt worden zonder gebruik te maken van de run-time kernel maar door gebruik te maken van de karakteristieken van de directe implementatie. Omdat het gedrag van de reflectieve toren volledige gedefinieerd is door de CCSR, kan de efficientie van het reflectieve programma gemakkelijk voorspeld worden. Tevens kan hij het gedrag van deze toren ook aanpassen en uitbreiden.

- Het metasysteem van de RbCl (dat is de CCSR) werd ontwikkeld gebaseerd op een object georienteerde en gelayerde architectuur. Zo kan de gebruiker zijn gedrag gemakkelijk aanpassen en uitbreiden in een ge-encapsuleerde manier.

- De reflectieve faciliteiten van RbCl worden geimplementeerd door een simpel mechanisme van de taal waarin geimplementeerd wordt, nl het co-routine mechanisme. De implementatie technieken kunnen gemakkelijk toegepast worden op een brede waaier van talen.

Daar RbCl een kernel-less systeem is kunnen alle features die hierboven beschreven werden, aangepast worden aan de noden van de gebruiker. We



Figuur 19: Een systeem met en zonder een runtime kernel

referreren naar de initiele taal als gewone RbCl om verwisseling te vermijden met aangepaste/uitgebreide RbCl.

5.1 Kernel-less systeem

RbCl is een kernel-less systeem. In dit deeltje beschrijven we de voordelen van zo een systeem.

Indien een systeem, geschreven in een programmeertaal L, de mogelijkheid biedt om alle faciliteiten te vervangen door gebruikers gedefinieerde code, dan spreken we van een kernel-less systeem op de taal L.

Volgende voorbeelden zijn voorbeelden van kernelloze systemen : wanneer de volledige machine-taal instructie code van een systeem opgeslagen is op een aanpasbare plaats, en het systeem heeft de mogelijkheid om alle waarden van alle adressen te veranderen, dan spreken we van een kernel-less systeem op de machine taal. Het windows systeem op een LISP machine is een kernel-less systeem op LISP, want alle LISP functies die het gedrag van de windows bepalen kunnen geherdefinieerd worden door de gebruiker.

Een beter uitgewerkt voorbeeld is het volgende : het gedrag van een reflectieve taal wordt meestal gedefinieerd door zijn reflectieve toren, dit is een oneindige toren van metacirculaire interpreters. Meer specifiek, het gedrag

van de taal op level n wordt gedefinieerd door de interpreter op level n+1. De gebruiker kan de volledige interpreter code op level n vervangen door gebruik te maken van de reflectieve computation op level n+1. Daarom is een systeem op level n een kernel-less systeem op de taal gedefinieerd in level n+1. Dit is een reden waarom de reflectieve systemen zo aanpasbaar zijn.

De meeste van de traditionele talen zoals LISP, PROLOG, SMALLTALK, etc., hebben een runtime kernel nodig om hun hogere level faciliteiten te ondersteunen.

RbCl is een kernel-less systeem. Dit kernel-less systeem werd gerealiseerd door een simpel mechanisme, Object Table genaamd. Hierdoor kan de gebruiker alles herdefinieren, de gehele C++ programma code[4] om zo het gedrag te veranderen tot op de restrictie van het Operating System en de hardware. Dus concurrente execution, inter-node communication, program code management, memory management, en zelfs de reflectieve schema's en faciliteiten zelf kunnen aangepast worden door de gebruiker. In de normale talen wordt de trade off tussen de verschillende karakteristieken zoals efficientie, flexibiliteit, veiligheid, portabliliteit, etc. beslist door de systeem programmeur. In RbCl kan deze balans van trade-offs ook door de gebruiker gewijzigd worden.

Een probleem met kernelloze systemen is meestal de moeilijkheid en het gevaar om het gedrag van zo een systeem dynamisch te veranderen. Daarom kan zo een systeem niet praktisch gebruikt worden. In RbCl laten deze reflectieve eigenschappen en de linguistic symbiose toe dat de gebruiker gemakkelijk het systeem kan veranderen, dit mede dankzij de encapsulatie die voorzien werd door de object-georienteerde natuur van RbCl.

5.2 Design en implementatie van het RbCl Metasysteem

Hier gaan we even dieper in op het design en de implementatie van het RbCl metasysteem en daarbijhorend beschrijven we de belangrijkste facilities die door RbCl gegeven worden.

5.2.1 Linguistic symbiosis

RbCl voorziet een nieuwe faciliteit en dat is linguistic symbiosis met C++ objecten. Het verbergt de implementatie 'gap' tussen RbCl objecten en C++ objecten door het toelaten van transparante inter-communicatie in dezelfde memory space. Een RbCl object kan dan een C++ object zien als een RbCl

[4]RbCl is geschreven in C++

object en omgekeerd. De gebruiker moet geen wetenschap hebben van de verschillen tussen deze 2 talen. Wanneer we communiceren tussen C++ objecten en RbCl objecten, dan gebruikt elk zijn eigen communicatie protocol : een RbCl object communiceert met een C++ object door middel van het RbCl message passing protocol en een C++ object communiceert met een RbCl object via de C++ virtuele functie invocatie. Het implementatie schema van de linguistic symbiosis wordt beschreven in 5.2.5. Linguistic symbiosis speelt, zoals beschreven zal worden in 5.2.3, een belangrijke rol in het implementeren van de oneindige reflectieve toren met eindige computing resources zonder een run-time kernel.

5.2.2 Metasysteem

Het client RbCl programma blijft op het basislevel. Het metasysteem is een metalevel systeem dat de uitvoering van het basislevel RbCl programma realiseert. Het metasysteem bestaat uit metalevel objecten. Zo ook wordt de uitvoering van de metalevel objecten gerealiseerd door een meta meta systeem en zo door tot in het oneindige en vormt zo een oneindige reflectieve toren.

Het RbCl systeem bestaat uit nodes dewelke units zijn van resource sharing zoals CPU power of geheugen. Elke node heeft zijn eigen reflectieve toren. Indien een node wordt geimplementeerd op een gedeelde geheugen architectuur machine, dan kunnen meerdere threads gelijkertijd runnen in de node. Anders gesteld, het metasysteem wordt sequentieel uitgevoerd; in deze zaak, concurrente uitvoering van de baselevel objecten worden pseudo parallell. Bij het programmeren van het basislevel hoeft de gebruiker geen kennis te hebben van de gedistribueerde natuur van de architectuur. Hoewel, het design van het metalevel houdt wel rekening met node boundaries. In het metasysteem van de gewone RbCl zijn er geen inter-node referenties tussen metalevel objecten. De inter-node communicatie tussen baselevel objecten wordt gerealiseerd op het metalevel door gebruik te maken van system calls die worden aangeleverd door het Operating systeem.

Elke level van elke node heeft zijn eigen System Object Table, dewelke een belangrijke rol speelt in het realiseren van de relfectieve faciliteiten van de RbCl. Een System Object Table realiseert een name-space voor systeem objecten. De systeem objecten zijn belangrijke metalevel objecten die het basis gedrag van het baselevel bepalen, en worden opgeslagen in het System Object Table van het metalevel. Bijvoorbeeld, de metalevel objecten die het volledige gedrag bepalen van de baselevel zoals de schedulers, de active queu's en de netwerkdaemons, zijn allen systeem objecten. De gebruiker kan het gedrag van het basislevel veranderen op een node, door het element in

het System Object Table van het metalevel op die node te vervangen.

Alhoewel het basis RbCl metasysteem enkel uit C++ objecten bestaat, is het gedrag van elk object zorgvuldig gedesigned om zo onafhankelijk mogelijk te zijn van de C++ features. Bijvoorbeeld, globale variabelen en globale functies van C++ worden niet gebruikt, daarvoor worden systeem objecten gebruikt zodat we encapsulatie van globale data en operaties verkrijgen. C++ object creatie constructs worden niet gebruik (zoals new class.foo()), we gebruiken generators daarvoor. Bijvoorbeeld als een systeem object genaamd cons-generator een message ontvangt, zal het een metalevel object genereren dat een cons cell voorstelt.

De gebruiker kan ook elke systeem object vervangen door een eigen object dat zowel een C++ object of een RbCl object mag zijn. De gebruiker moet geen rekening houden met het verschil tussen deze 2 talen dankzij de linguistic symbiose. Als de gebruiker RbCl systeem objecten definieert dan kan de gebruiker alle RbCl features gebruiken die er voor handen zijn, zoals concurente uitvoering, inter-node communicatie etc.

Het RbCl metasysteem wordt geconstrueerd in een gelayerd model. Met andere woorden bestaat het metasysteem uit meerdere layers die overeen stemmen met de graad van abstracties. Programma modules die afhankelijk zijn van de hardware architectuur zijn in de lagere lagen en programma modules die van meer specifieke features afhankelijk zijn, zitten in een hogere laag. De gelaagde architectuur laat toe om portable te zijn en laat uitbreiding toe.

5.2.3 De oneindige toren van directe implementaties.

Zonder een run-time kernel zou het vorige reflectief systeem onmogelijk te implementeren zijn met eindige computing resources. Een runtime kernel zorgt er meestal voor dat het systeem reageert als een oneindige toren. Dit in tegenstelling tot het RbCl systeem dat een toren embodied, en geimplementeerd werd met een eindig aantal computing resources zonder een run-time kernel. Dit werd mogelijk gemaakt door de linguistic symbiosis met C++ objecten die kunnen uitgevoerd worden zonder runtime kernel. De reflectieve toren van de plain RbCl is een oneindige toren van directe implementatie. In deze sectie beschrijven we deze reflectieve toren van RbCl.

In een reflectief systeem bestaan geen baselevel entiteiten op het metalevel in de strikte zin van het woord. Dit wil zeggen dat er enkel metalevel entiteiten bestaan die baselevel entiteiten voorstellen. Bijvoorbeeld, stel u een LISP interpreter voor die geschreven is in C. Alhoewel er cons cellen zijn in LISP, zijn deze er toch niet in C. In C zijn er C-structuren die een LISP cons cel voorstellen. Op dezelfde manier zijn er geen C structuren op het machine

language level, maar er is eerder storage waarbij hun inhoud C structuren voorstellen. We kunnen zo ver gaan met zeggen dat er geen storage is op het hardware niveau, maar dat er eerder electronische entiteiten zijn die storage representeren. Wanneer dan een LISP programma runt zijn alle levels actief op hetzelfde moment.

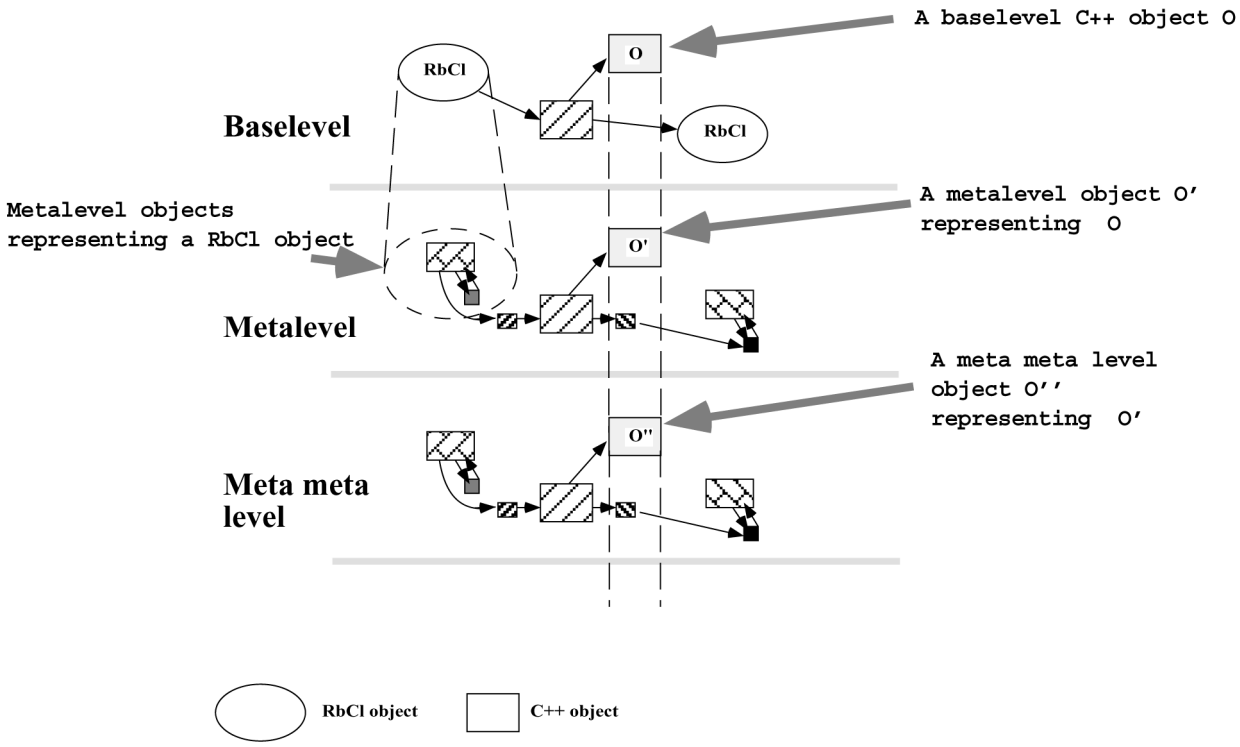
Laat ons even aandacht schenken aan het C level dat LISP expressies interpreteert. In het algemeen zijn er 2 mogelijkheden om een taal faciliteit te programmeren bovenop een andere taal : directe implementatie en expliciete implementatie. Bijvoorbeeld voor de + operatie van LISP werd gebruik gemaakt van de + operatie van C. We zeggen dat de + operatie direct is geimplementeerd. Zo zal de semantiek van de + operatie in LISP afhankelijk zijn van deze in C. Aan de andere kant, als we de betekenis van de + operatie expliciet gaan definieren gebruik makende van de primitieve recursie, dan kunnen we zeggen dat de + operatie expliciet geimplementeerd is.

Naar analogie, laat ons eens aandacht schenken aan het metalevel van RbCl dat het basislevel van RbCl implementeert. RbCl voorziet de linguistic symbiosis die toelaat om C++ objecten uit te voeren als baseclass objecten. In het gewone RbCl metasysteem zijn de uitvoering van de baselevel C++ objecten direct geimplementeerd. Dit werd mogelijk gemaakt als volgt : een baselevel C++ object O wordt voorgesteld door een single metalevel object O' (in tegenstelling tot een baselevel RbCl object hetwelk voorgesteld wordt door meerdere metalevel objecten). O' heeft instantie variabelen en methodes identiek aan deze van O. Daarom zal een message zenden naar O op het basislevel simpel voorgesteld worden als een message verzenden naar O' in het metalevel.

Laat ons ook even aandacht schenken aan het meta meta level van RbCl dat het metalevel van RbCl implementeert. Er zijn meta meta level C++ objecten die metalevel C++ objecten voorstellen. Bijvoorbeeld zal het metalevel object O' gerepresenteerd worden door een meta meta level object O" dat dan instance variabelen en methodes heeft identiek aan deze van O'. Omdat het plain RbCl metasysteem enkel uit C++ objecten bestaat, is de computation state van het meta meta level identiek aan dat van het metalevel. Op deze manier kan het meta meta level van de plain RbCl gezien worden als actueel bestaand en direct geimplementerende uitvoering van metalevel C++ objecten.

Het meta meta level zelf kan ook gezien worden als direct geimplementeerd door het meta meta meta level en deze toren van directe implementaties gaat oneindig voort. Op deze manier wordt de reflectieve toren van de RbCL gerealiseerd met een eindig aantal computing resources.

Overeenkomstig deze opvatting, willekeurige (niet reflectieve) systemen kunnen gezien worden als geimplementeerd door oneindige torens van directe



Figuur 20: De oneindige toren van directe implementatie

implementatie. Dergelijke visie is meestal zinsloos omdat de gebruiker van zo een systemen de lagere levels niet kan manipuleren. Aan de andere kant, de reflectieve faciliteiten van RbCL laten de gebruiker toe om willekeurige levels aan te passen aan de noden van de gebruiker. Het meta meta systeem dat de RbCL objecten interpreteert op het metalevel is expliciet gegenereerd door het metasysteem wanneer een gebruiker voor de eerste keer een RbCL object creeert in het metasysteem. Daarom heeft de oneindige toren van directe implementatie van de RbCL dezelfde mogelijkheden als de reflectieve toren van andere reflectieve systemen. Hoe het meta meta systeem gegenereerd wordt, wordt beschreven in 5.2.6.

5.2.4 Level Shifting door level managers

De linguistic symbiosis laat toe dat C++ objecten worden uitgevoerd in het baselevel. Zoals uitgelegd in 5.2.2 heeft elke level zijn eigen System Object Table dat de name space van het systeem object realiseert. Daaruit volgt dat de baselevel C++ objecten moeten uitgevoerd worden in de baselevel name space. Het management van deze name spaces wordt geleid door een systeem object, level manager genaamd. Elke level van elke node heeft zijn eigen level manager. De level manager van het metalevel voert level shifting uit en dat wil zeggen het switchen van de huidige name space van het systeem object tussen het basislevel en het metalevel. De levelmanager van het meta meta level voert gelijkende level shifting uit tussen het metalevel en het meta meta level. De linguistic symbiosis en al de reflectieve faciliteiten zoals het het creeeren van de metalevel RbCl objecten wordt geimplementeerd gebruik makende van de primitive level shifting mogelijkheid gegeven door de level managers. Merk op dat een level manager enkel de name spaces switched. Wanneer de name space van het metalevel geshift wordt, wordt de baselevel nog steeds uitgevoerd door het metasysteem en het metalevel wordt nog steeds uitgevoerd door het meta meta systeem enzovoort.

Het level shifting mechanisme wordt als volgt geimplementeerd : een C++ object refereert naar een C++ globale variabele om de juiste system-object-table die de huidige name space representeert, te kennen. Een C++ object accesses een systeem object als volgt :

```
system_object_table[symbol_id]
```

System_object_table is een globale C++ variabele dewelke een pointer is naar een C++ array van objecten. Deze array stelt een system object table voor. Symbol_id is een small integer die een globale naam van een systeem object voorstelt. De level manager verandert de waarde van de globale C++

variabele system_object_table naar de juist system-object-table wanneer het een message :shift-to-meta of : shift-to-base toegestuurd krijgt.

Level managers zijn systeem objecten en kunnen dus ook deze vervangen worden door objecten gedefinieerd door de gebruiker. Daarom kunnen programma's die veranderingen aan de reflectieve faciliteiten van de RbCl zelf, zoals debugging van de reflectieve programma's, of het uitvoeren van experimenten op de reflectieve faciliteiten, kunnen beschreven worden in het RbCl talen framework.

5.2.5 Implementatie van de Linguistic Symbiosis

Om de linguistic symbiosis te realiseren moet de implementatie gap tussen RbCl en C++ geabsorbeerd worden door metalevel objecten genaamd intermediate pointers, dewelke instaan voor 1) conversie van het message passing protocol tussen de 2 talen, en 2) het level shifting door het zenden van messages naar de level manager of het metalevel. Elke referentie tussen baselevel Rbcl objecten en baselevel C++ objecten wordt voorgesteld als een intermediate pointer object op het metalevel. De gebruiker kan het gedrag van deze intermediate pointer aanpassen en/of uitbreiden door gebruik te maken van de reflectieve eigenschappen van de taal.

Een intermediate pointer managed ook de coroutine faciliteit door gebruik te maken van de thread library van de C taal. Alle baselevel RbCl objecten worden geinterpreteerd in het metalevel door 1 thread en baselevel C++ objecten worden direct geimplementeerd door metalevel C++ objecten die runnen op het metalevel op een andere thread. Wanneer een baselevel RbCl object een message verstuurt naar een baselevel C++ object dan zal de intermediate pointer van het metalevel de active thread naar een nieuwe thread switchen waar het corresponderende metalevel C++ object runt. Als de controle wordt teruggegeven naar het baselevel RbCl object, dan wordt de interpreter thread van het metalevel terug geactiveerd. Message passing tussen 2 C++ objecten wordt efficient afgehandeld door een gewone virtuele functie invocatie zonder gebruik te maken van intermediate pointers.

5.2.6 Generatie van het meta meta systeem

Vele reflectieve systemen implementeren lazy creation van het metasysteem in een runtime kernel. Het RbCl metasysteem kan het meta meta systeem genereren in het RbCl taal framework - dit wil zeggen dat het geen runtime kernel nodig heeft om dit te doen. Dit werd als volgt verkregen : laat ons eerst een object definieeren dat een primitieve is enkel als het geimplementeerd is door een directe implementatie. Bijvoorbeeld zijn C++ objecten primitieve

objecten terwijl geinterpreteerde RbCl dit niet zijn. Zoals uitgelegd in 5.2.3 , wanneer een primitief object runt in het metalevel dan zal hetzelfde primitieve object eigenlijk in het meta n+1 level runnen. Bijvoorbeeld, wanneer een metalevel primitief object A een ander primitief metalevel object creeert en deze zend ook een message naar B, dan zal het overeenkomstige meta meta level object A' eigenlijk het overeenkomstige meta meta level object B' creeeren en de message naar B' zenden.

Het meta meta systeem kan gegenereerd worden door het meta systeem gebruik makende van de eigenschappen van primitieve objecten. Om het kort samen te vatten, het enige dat moet gedaan worden is een nieuw metasysteem genereren dat enkel uit primitieve objecten bestaat.

Het meta meta systeem werd door het metasysteem op volgende wijze gegenereerd :

1. Er wordt een array van objecten gecreeerd die het System Object Table voorstelt in het meta meta systeem

2. Er wordt een "Shallow-copy" van alle elementen van het default-system-object-table naar het system-object-table van het meta meta systeem gedaan. De default-system-object-table is een systeem object dat alle systeem objecten van het plain RbCl metasysteem bevat die geshared worden door alle levels. De meeste objecten zoals generators hebben geen interne staat zodat ze kunnen geshared worden.

3. Het creeeren van additionele objecten en hen registreren in de system-object-table. Deze objecten zijn systeem objecten en kunnen niet geshared worden over alle levels. Bijvoorbeeld een level manager en een scheduler.

4. Initialiseer de gecreeerde systeem objecten alsof ze in de uitvoering van het metalevel direct geimplementeerd waren door het meta meta systeem.

Het gegenereerde meta meta systeem voorziet alle faciliteiten die door het plain RbCl voorzien werden. Wanneer een RbCl object wordt gecreeerd in het meta meta level, dan zal het meta meta meta level op dezelfde manier gegenereerd worden.

5.3 Conclusie

RbCl realiseert de reflectieve toren efficient met een eindig aantal computing resources en zonder een run-time kernel. Dit wordt bereikt door gebruik te

maken van een simpel mechanisme, system object tables genaamd en een nieuwe faciliteit nl. linguistic symbiosis met C++ objecten.

Omdat RbCl een kernel-less systeem is kan de gebruiker het gedrag van het systeem veranderen tot op de restrictie van het OS en de hardware binnen het RbCl framework. Dit impliceert dat elke mogelijke run-time faciliteit kan voorzien worden als een library of applicaties geschreven in RbCl.

Het System Object Table van het metalevel op een node is de CCSR van het baselevel van de node. Het systeem object dat het basis gedrag van het baselevel realiseert zijn de elementen in het system object table van het metalevel. De voordelen van het system object table zijn de volgende :

1. Het system object table van het metalevel is het mechanisme dat het RbCl metasysteem kernel-less maakt. Al de system objecten worden indirect ge-accessed door een system object table. Daardoor kan de gebruiker het gedrag van het metasysteem veranderen door het vervangen van elementen van het system object table door zelf gedefinieerde objecten. De overhead door deze indirectie is verwaarloosbaar omdat toegang tot systeem objecten in enkele instructie-stappen kan gedaan worden.

2. Het system-object-table representeert de name-space van system objecten die gelijk worden geaccessed door verschillende talen zoals C++ en RbCl.

3. Level shifting, het switchen tussen de huidige namespace van de system objecten, kan zeer efficient geimplementeerd worden.

De voordelen van linguistic symbiosis met C++ zijn de volgende :

1. Een efficient reflectief systeem kan gemakkelijk geimplementeerd worden. Het metasysteem kan geconstrueerd worden enkel door de C++ objecten die efficient worden uitgevoerd, en de gebruiker kan de metalevel C++ objecten manipuleren zoals gewone RbCl objecten.

2. De linguistic symbiose dient ook als een vreemde taal interface naar de C(++) objecten die de gebruikers toelaten om system calls, die voorzien worden door het OS, te gebruiken.

Het moet opgemerkt worden dat de implementatie-technieken gebruikt in RbCl gemakkelijk gebruikt kunnen worden in andere systemen. De reflectieve faciliteiten van RbCl worden geimplementeerd, gebruik makende van simpele mechanismen van de implementatie taal : het system object table is enkel een simpele array van objecten, en de linguistic symbiosis heeft enkel nood aan de coroutine faciliteit voor zijn implementatie.

6 Virtual Virtual Machine

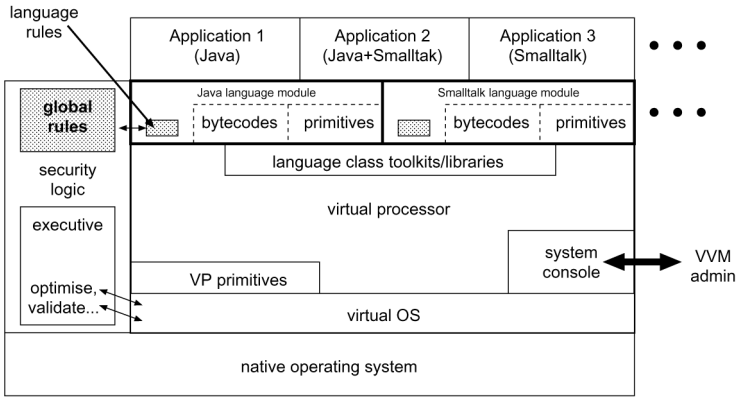
> A philosopher is a person who doesn't care which side his bread is butterdone; he knows he eats both sides anyway! – **Joyce Brothers**

Nu bespreken we de 'Virtual virtual machine' of ook wel VVM genaamd. De virtuele virtuele machine (VVM) is een multi-language, hardware onafhankelijk execution platvorm dat dynamisch uitbreidbaar en aanpasbaar is aan iedere nood van een applicatie. Deze nieuwe virtuele machine laat toe dat programma's geschreven in een willekeurige bytecode taal, uitgevoerd worden in veilige en efficiente omgeving. Applicaties kunnen geschreven worden in de meest bruikbare taal en de omgeving kan aangepast worden aan hun specifieke noden.

6.1 De VVM Architectuur

Virtuele omgevingen en hogere orde programmeertalen verbeteren de kwaliteit van de software en zorgen tevens voor een mindere kost bij de ontwikkeling van applicaties. Maar de aanvaarding van virtuele machines was klein omdat deze nogal traag waren. De laatste tijd werden de virtuele machines al heel wat sneller. Een ander nadeel was ook de onaanpasbaarheid van de virtuele machines. De VVM daarentegen kan geconfigureerd worden om de applicatie, geschreven in eender welke bytecode, uit te voeren. Het maakt van de VMM een flexibele omgeving. VVM applicaties zijn "getypeerd" met een overeenkomstig "execution model". Elke getypeerde applicatie komt overeen met een VM beschrijving (een VMlet). VMlets worden op vraag geladen, telkens een nieuwe applicatie type wordt tegengekomen. Deze VMlets bestaan uit regels die beschrijven hoe de applicatie moet vertaald worden naar de interne representatie gebruikt door de VMM.

- **Virtual Processor.** De core-machine is de virtuele processor, dewelke een low-level engine voorziet. Deze engine heeft een simpele intern object-model (en bijhorend object geheugen), een instructie set en een instructie model. Het gebruikt een taal-neutrale, interne representatie die een goede performantie heeft, een veel kleinere 'footprint' dan dynamisch vertaalde native code en een hoge graad van portabiliteit. De innerlijke VP instructies zijn elementair. Ze voorzien de bouwstenen die nodig zijn om een brede waaier aan bytecode-virtuele-machines te implementeren. De VP interageert met het virtuele operating systeem (VOS), dewelke een abstractie creeert over de voorzieningen van het native operating system.



Figuur 21: De VVM architectuur

- **Virtual Operating System.** VOS services worden verkregen door VP programma's door het oproepen van virtuele processor primitieven. Bijvoorbeeld het Virtual Operating Systems (VOS) moet de VP scheduling en het managen van beveiligde resources beheren. Het is ook verantwoordelijk voor het herconfigureren wanneer een nieuwe VMlet of toolkit wordt ingeladen.

- **Language Toolkits.** Deze worden aangeleverd in de vorm van instructie bibliotheken. Ze voorzien functies die bij een familie van talen hoort, in de vorm van nieuwe VP instructies, gedefinieerd door bestaande VP instructies (op dezelfde manier waar elke VMlet een nieuwe VP instructie voor elk van zijn bytecodes definieert). Typische VP bibliotheken voorzien cloning, delegatie of inheritance mechanismen en speciale data representatie zoals getagde "onmiddellijke" types.

- **VMlets.** Ondersteuning voor elke specifieke taal binnen een familie worden voorzien door VMlets. Elke VMlets heeft 5 dingen : 1) een mapping van de VM bytecode naar de VVM's execution mechanisme. 2) Een mapping van de VM objecten naar het VVM's object formaat. 3) Een implementatie van de VM's primitieven. 4) Een lader voor de VM's applicatie en uiteindelijk nog 5) een type checking specificatie en dynamische security regels voor een gegeven taal. Elke VM bytecode komt overeen met een enkele, mogelijke nieuwe, VP instructie. Deze nieuwe VP instructies worden gedefinieerd wanneer de VMlets wordt ingeladen, in termen van reeds bestaande VP instructies.

- **Security Module.** Het VOS werkt samen met de security module om

zo de "management" taken te vervullen die nodig zijn om de integriteit van de operaties te waarborgen. De globale (inter-talen) aspecten van deze taken worden beschreven door een verzameling van regels in de security module zelf. Regels die verwant zijn met individuele talen worden voorzien door elke VMlet. Deze regels beschrijven de security constraints voor de applicaties die werken op de VM, en hoe de VP de objecten mag manipuleren die bij deze VM behoren.

- **Application programs.** De applicatie bytecodes van elke taal worden dynamisch vertaald in VP uitvoerbare instructie voorstellingen. Daar alle applicaties worden vertaald in 1 enkel uitvoerbaar formaat (onafhankelijk van welke VM hun semantiek defineert), wordt een enkel uitvoerings mechanisme (gedefineerd door de VVM) gebruikt bij de uitvoering van de code.

6.2 VVM Werking

Deze sectie vat de operaties van de VVM samen, vanaf de aankomst van een applicatie op de VVM tot zijn uitvoering.

Wanneer een applicatie toekomt op de VVM wordt zijn type gechecked. Als de VVM nog niet de nodige VMlets geladen heeft, dan wordt deze gelocaliseerd en ingeladen. De security module verifieert de VMlets overeenkomstig de globale regels, om er zich van te vergewissen dat de integriteit van de VVM niet beschadigd wordt. De VP instructie set wordt uitgebreid met operaties gedefinieerd door de VMlet of geimporteerd uit een of meerdere VP libraries die over de gemeenschappelijke operaties voor een familie van talen, waartoe de VM behoort, beschikt.

Daarna wordt de applicatie dan ingeladen. Taal-specifieke security regels, gedefinieerd door de VMlet, worden toegepast op de applicaties bytecode en data. De data van de applicatie worden dan vertaald naar de VVM's interne representatie.

De uitvoering van de applicatie kan nu beginnen. Methodes worden dynamisch vertaald in de overeenkomstige VP instructie voor de uitvoering. De VMlet voorziet primitieven voor zijn applicaite dewelke worden aangesproken op een manier overeenkomstig de taal. Deze kunnen eventueel VOS services en interne primitieven gedefinieerd door de VP, starten.

6.3 Conclusie

In dit hoofdstuk hebben we de virtuele virtuele machine eens van wat dichter bekeken. Virtuele virtuele machines zijn eigenlijk reflectieve machines. Ze

definieren een virtuele machine boven een andere virtuele machine, zodat deze eerste virtuele machine aanpasbaar is. Dit is eigenlijk het idee van de oneindige toren van interpreters zoals gezien bij de reflectieve talen. De nadruk ligt in deze machine duidelijk op aanpasbaarheid en niet op performantie. Het gaat er van uit dat de computers snel genoeg moeten zijn om een aanvaardbare snelheid te krijgen voor deze machine. Maar een toren van interpreters is wel heel traag en daar kunnen nog verbeteringen aan gedaan worden.

7 NitrO

7.1 Overzicht

NitrO is een Virtuele machine die ontwikkeld werd aan de universiteit van Oviedo in Spanje.

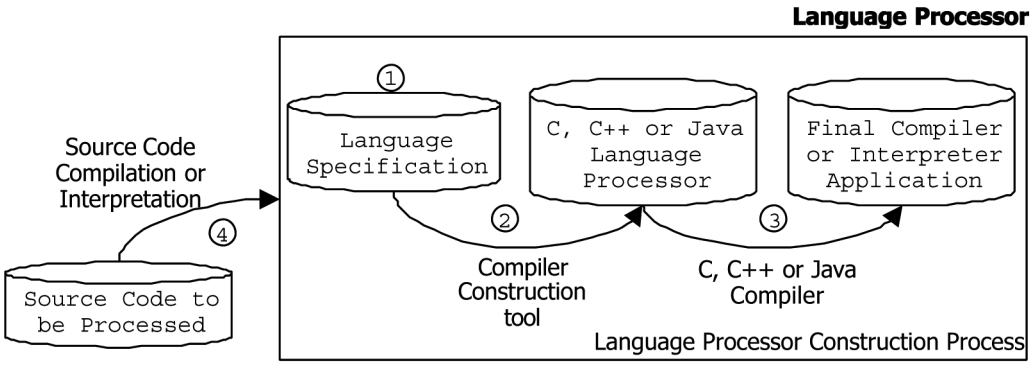
NitrO heeft volgende main features :

Prototype-based object model :

- De basis abstractie is het object. Men gebruikt een prototype object model in welke klassen niet bestaan.
- De virtuele machine gebruikt een simpel object model zodat de klasse abstractie niet nodig is.
- Er is geen verlies van representatie door gebruik te maken van een prototype-gebaseerd object model in plaats van gebruik te maken van een klasse-gebaseerd model.
- Verschillende object-georienteerde talen kunnen gemakkelijk vertaald worden tot een prototype gebaseerde object-georienteerde taal.
- In een klasse-gebaseerd model moet om de structuur van een object te veranderen, de gehele klasse veranderd worden. Maar wat gebeurt er met de andere objecten van de klassen? MetaX creeert een nieuwe klasse, shadow klasse genaamd, enkel voor dit object, zodat het object model complexer en moeilijker te implementeren is. In een prototype-gebaseerd model wordt dit gemakkelijk bereikt door middel van object cloning en dynamische inheritance.

Structurele Reflectie :

- Een object wordt gedefineerd als een verzameling van referenties naar andere objecten.
- Er zijn 2 primitieve objecten: het nil-object en het string-object.
- Elk object biedt structurele-reflectieve operaties (delete, insert en iteratie over elk element van de referentie verzameling) aan.
- Het gedrag wordt voorgesteld door middel van string objecten; deze objecten kunnen dynamische gecreeerd, gemanipuleerd en geevalueerd worden met de () operator. Een methode wordt gedefinieerd als een evalueerbare string die een member is van het object.



Figuur 22: Development process

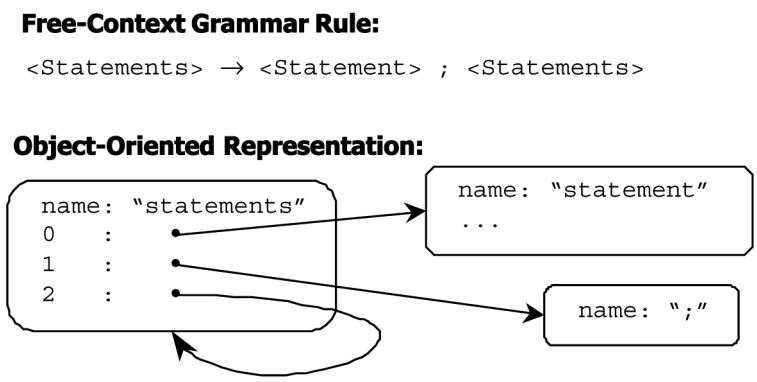
Uitbreidbaarheid (Extensibility) : Het computational model van de virtuele machine werd zo gedesigned dat het zeer klein is. Door middel van structurele reflectie werd een programmeer-omgeving gecreeerd boven op de basis van de virtuele machine (dit is platvorm-onafhankelijke en heeft portable code). Deze programmeer-omgeving behaalt een hoger abstractie level : aanpasbare persistentie, distributie en thread scheduling systemen werden ontwikkeld in deze programmeeromgeving.

En men ondersteunt applications interoperability : Juist één virtuele machine bestaat er voor elke fysische computer. In tegenstelling tot de Java virtuele machine zullen verschillende applicaties op dezelfde fysische computer, eenzelfde nitrOVM gebruiken. Het voordeel hiervan is dat elke applicatie de mogelijkheid heeft om elk object in de virtuele machine (niet enkel zijn eigen objecten) te kunnen gebruiken door middel van structurele reflectie.

7.2 Non-restrictive Computational Reflection System

De meeste compilers en interpreter-tools (van een klassieke 'lex and yacc' tot moderne Java) hebben allemaal hetzelfde development process. figuur 22

1. De taal-specificatie wordt gedefinieerd.
2. De taal-specificatie wordt gepreprocessed en een C, C++ of Java applicatie wordt gegenereerd.
3. Eenmaal de applicatie gegenereerd is, wordt het gecompileerd om zo de language processor te verkijgen.



Figuur 23: Free-context grammar rule

4. Met de gecreeerde compiler of interpreter kunnen we source code van de verlangde taal processen.

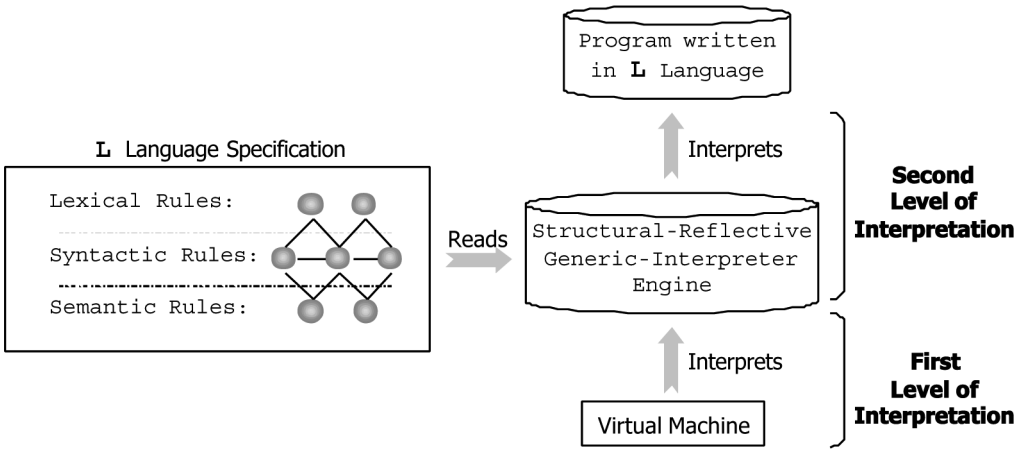
Als we nu een taalfeature willen aanpassen, dan moet het gehele process herhaald worden. Daarom is er geen dynamische modificatie mogelijk van de geprocessde taal door middel van dit model.

De nitrO language processor is gedefinieerd als een 'Structural-Reflective Generic Interpreter" (SRGI). Lexicale en syntactische specificaties worden voorgesteld door objecten wat wil zeggen "free-context" grammatica regels. Het object representeert de linker zijde van de regel en de rechter zijde wordt voorgesteld door de 'member' verzameling van het object. figuur 21. De regels mogen worden gecreeerd, geanalyseerd en dynamisch aangepast door gebruik te maken van de structurele reflectie van de virtuele machine.

Semantische specificaties geassocieerd met syntactische regels worden beschreven door string objecten. Deze kunnen gemakkelijk geevalueerd worden door de virtuele machine door middel van de () operator.

De SRGI engine werd gedesigned om een backtracking algoritme te gebruiken. Eenmaal de taal die moet geinterpreteerd worden, gespecifieerd is, start de SRGI engine deze te processen. Het volgt daarbij een top-down schema. Het resultaat is een twee-level interpreter toren : eerst de virtuele machine en ten tweede de geinterpreteerde taal door de SRGI engine.

De SRGI kan elke taal interpreteren -we zullen deze L noemen- gebaseerd op zijn specificatie. En het zal altijd één taal kunnen interpreteren zonder een specificatie, nl. de nitrOVM taal. Gebruik makende van het gereserveerde woord 'reflect' mag er nitrOVM code gebruikt worden. De SRGI engine neemt deze code aan als een string object en evalueert deze met de ()-operator. Dus de eerste level (nitrOVM) evalueert de code geschreven binnen de 'reflect' statement in plaats van de tweede level (SRGI). 1 level van



Figuur 24: De structuur van de 2 levels in generische interpretatie

interpretatie werd zo overgeslagen.

Als de gereflecteerde nitrOVM code nu de L-code aanpast door middel van de nitrOVM structurele reflectie, wat we nu bereiken is een non-restrictief computationeel reflectie mechanisme. Met dit schema wordt de nitrOVM taal een meta-taal die de taal specifieert, en dynamisch veranderd, die we zouden interpreteren. Dit zonder gebruik te maken van een MOP die specifieert wat er mag veranderd worden.

7.3 conclusie

NitrO is een reflectieve virtuele machine waarbij het gedrag van nitrO at run-time kan aangepast worden. Elke feature wordt gereflecteerd : introspectie, structurele reflectie, computationele reflectie en "linguistic" reflectie werden behaald en dit zonder gebruik te hoeven maken van een MOP (meta object protocol). Maar ook hier moeten we voor deze runtime reflectie betalen met performantie verlies door dubbele interpretatie.

8 Black

"Black is Black" – **Los Bravos**

8.1 Inleiding

'Black' is een LISP gebaseerde reflectieve taal die toelaat om at runtime wijzigingen aan te brengen aan de taal. Daar waar conventionele LISP gebaseerde reflectieve talen, zoals 'Brown' en 'Blond', de oneindige toren van interpreters op een lazy manier construeert door middel van direct uitvoerbare interpreters in plaats van geinterpreteerde interpreters, is 'Black' een direct uitvoerbare interpreter geimplementeerd door duplicatie en partiele evaluatie. En hierbij wordt de aanpasbaarheid behouden.

8.2 De oneindige toren

De syntax van Black is dezelfde als Scheme maar heeft één extra reflectieve construct, 'exec-at-metalevel E'. Zoals in vele andere reflectieve talen bestaat er in Black ook een oneindige toren van interpreters. De construct 'exec-at-metalevel E' wordt gebruikt om de expressie E een level hoger te evalueren. De Black interpreter is een gewone CPS (continuation passing style) Scheme interpreter, behalve voor de functie eval-EatM$_n$[5]

eval-EatM$_n$[6] is gedefinieerd om de exec-at-metalevel construct te evalueren. Door het oproepen van eval$_{n+1}$ wordt de body E van 'exec-at-metalevel E' geevalueerd op level n. Dit wil zeggen op hetzelfde level als waar eval-EatM$_n$ runt. Laat ons deze interpreter I$_n$(.) noemen. Een Black programma exp wordt dan conceptueel geinterpreteerd door een oneindige toren van I$_n$(.) als volgt :

... I$_3$(I$_2$(I$_1$(exp, env$_1$),env$_2$),env$_3$)....

Elke I$_n$ definieert een operationele semantiek van de taal L$_{n-1}$ van level n-1 en interpreteert I$_{n-1}$ geschreven in L$_{n-1}$. Op het laagste niveau van de toren interpreteert I$_1$ gebruikers programma's. Merk op dat I$_n$ niet altijd geinterpreteerd wordt door I$_{n+1}$ omdat in eval-EatM$_n$, I$_n$ een vrije variabele env$_{n+1}$ bevat en een oproep naar eval$_{n+1}$(I$_{n+1}$) die niet werden gedefinieerd in I$_n$. In plaats van geinterpreteerd te worden door I$_{n+1}$ roept dit deel direkt I$_{n+1}$ op om de body van exec-at-metalevel expressie 1 level hoger uit te voeren. We kunnen dus zeggen dat eval-EatM$_n$ een 'hole' bevat naar hogere levels dewelke niet kan verklaard worden uit de gewone toren structuur.

[5]EatM is een afkorting voor Eval-at-Metalevel.

[6]Subscript aan functie namen geven de level van de functie aan.

8.3 Implementatie Moeilijkheden

De oneindige toren van interpreters zoals hierboven beschreven is onmogelijk te programmeren, daar alle interpreters geinterpreteerd dienen te worden door een interpreter die zich 1 level hoger bevindt, en zo een oneindige regressie veroorzaakt. Jefferson en Friedman implementeerden een eindige toren van interpreters door gewoon het aantal interpreters te limiteren. Alhoewel we in de praktijk maar enkele levels nodig hebben is deze oplossing verre van efficientie. Daar elke interpreter moet geinterpreteerd worden door een interpreter 1 level hoger, wordt dit systeem ongelofelijk traag.

Brown en Blond maakten de oneindige toren van interpreters efficient door de direct uitgevoerde interpreters op een luie manier te creeeren. Deze methode laat ons spijtig genoeg niet toe om de interpreters zelf te herdefinieren omdat deze direct werden uitgevoerd in machine-taal. We kunnen wel vrij op en neer gaan in de toren van interpreters en daardoor omgevingen en continuaties naar believen aanpassen enzovoort, maar we kunnen niet de interpreters zelf wijzigen. Alhoewel de direct implementatie noodzakelijk is voor efficientie van reflectieve talen, kan het niet toegepast worden op Black omdat bij Black ook de interpreters aanpasbaar moeten blijven.

8.4 Performante Interpretatie

Zoals reeds gezien in punt 2.6 heeft compilatie tot gevolg dat de uitvoering van een programma zeer snel gebeurd (sneller dan interpretatie) maar (there's no such thing as a free ride) het programma is niet meer aanpasbaar. Interpretatie laat wel toe om at runtime aanpassingen te doen maar de kostprijs hiervoor is efficientie. Het is mogelijk om een performante interpreter te creeeren als we de mogelijke aanpassingen beperken.

In Black zetten we de beperking dat enkel aanpassingen in de metalevel omgeving het gedrag van de baselevel gecompileerde code zal wijzigen. Dit is, als de metalevel omgeving verandert, gebruik makende van de set!-operator, dan zal het gedrag van de gecompileerde code op het baselevel gewijzigd worden. Maar als we functies in de metalevel interpreter herdefinieren, dan zal de baselevel gecompileerde code geen wijziging in gedrag vertonen. Dit wil zeggen dat gecompileerde code correct uit de voeten kan met een programma die enkele van zijn functies herdefinieert tijdens zijn uitvoering, maar het niet overweg kan met een programma dat zijn metalevel interpreter wijzigt. Men vond deze restrictie aanvaardbaar omdat we elke gecompileerde functie kunnen veranderen door een zelf-gedefinieerde functie door deze in de omgeving aan te passen gebruik makende van set!. We kunnen dus de gecompileerde metalevel interpreter aanpassen om zo het gedrag van de baselevel geinterpre-

$$
\begin{array}{ll}
\ldots & \ldots \\
L_3 : \mathtt{I}_3(exp, env_3) & S : \mathcal{I}_3(exp, env_3, env_4) \\
L_2 : \mathtt{I}_2(exp, env_2) & S : \mathcal{I}_2(exp, env_2, env_3) \\
L_1 : \mathtt{I}_1(exp, env_1) & S : \mathcal{I}_1(exp, env_1, env_2) \\
L_0 : \mathtt{P}(arg) & S : \mathcal{P}(arg, env_1)
\end{array}
$$

Figuur 25: Overzicht van Black

teerde programma's aan te passen. We kunnen niet het gedrag van baselevel gecompileerde programma's aanpassen, maar dit is vanzelfsprekend want ze zijn gecompileerd.

8.5 De structuur van de Black interpreter

We zullen u even de structuur van de Black interpreter laten zien. In figuur 25 stelt de linker kolom het overzicht vanuit het standpunt van de gebruiker voor. L_n : I_n (exp,env_n) stelt dat de interpreter I_n geschreven is in de taal L_n. Het definieert de taal L_{n-1} en interpreteert I_{n-1}. Initieel zijn alle L_n dezelfde taal, maar deze kunnen verandert worden als we de taal semantiek aanpassen gebruik makende van exec-at-metalevel. De rechter kolom in de figuur stelt de direct geimplementeerde interpreter voor die gebruikt wordt om Black programma's te interpreteren. S : I_n (exp, env_n, env_{n+1}) wil zeggen dat de interpreter I_n geschreven is in de taal S. S is de taal die gebruikt werd om de direct uitvoerbare interpreters te implementeren. In het geval van Black is dit Scheme.

Een niet gecompileerd gebruikers programma P wordt meestal geinterpreteerd door I_1. Als (exec-at-metalevel E) wordt opgeroepen in P dan roept $I1$ $I2$ op om de body van E te interpreteren in het metalevel. Stel u voor dat een set! statement in E een functie van $I1$ herdefinieert dan geeft dit een verandering in env_2 in I_2. Daar env_2 gedeeld wordt door I_2 en I_1 kan I_1 een verandering waarnemen en de geherdefinieerde functie gaan gebruiken. I_1 is dus gewijzigd. Het delen van de omgevingen tussen verschillende levels is belangrijk hier. Door het delen van omgevingen zullen veranderingen aangebracht in I_2, het gedrag van I_1 wijzigen. Nadat I_1 werd gewijzigd zullen gebruikersprogramma's P vanaf dit moment geinterpreteerd worden door de aangepaste I_1.

8.6 Construeren van de default I_n

We construeren I_1 maar de andere I_n zijn analoog van opbouw. De uiteindelijke uitkomst van I_1 lijkt nogal zeer op die van I_1 met het verschil dat I_1 ook nog toegang heeft tot env_2, wat eigenlijk zou gebeuren indien I_1 geinterpreteerd zou zijn door I_2. Met andere woorden is I_1 het resultaat van de specialisatie van I_2 op I_1, waarbij de toegang tot env_2 behouden blijft :

$$I_1(\text{exp},\text{env}_1,\text{env}_2) = \text{PE}(\text{I}_2(\text{I}_1(\text{exp},\text{env}_1)\text{env}_2))$$

Hier stelt PE een partiele evaluator voor. We partieel evalueren I_2 op I_1 met de onbekende parameters exp, env_1, en env_2. Algemeen gezien is het resultaat van deze partiele evaluatie I_1 zelf omdat I_2 een meta-circulaire interpreter is.

8.7 Compileren d.m.v. een partiele evaluator

Black gebruikt ook een partiele evaluator als compiler om zowel gebruikersprogramma's als voor interpreters. Alhoewel de interpreter uit het vorige punt redelijk efficient lijkt te zijn, begint deze na een tijdje trager te worden doordat (er delen van) de interpreter geherdefinieerd zijn. Bijvoorbeeld als we de volledige I_1 vervangen door een gebruikers gedefinieerde I'_1 dan zal het programma P geinterpreteerd worden door I'_1 welke op zijn beurt zal geinterpreteerd worden door I_2, hetgeen nogal inefficient is. Om deze interpretatie overhead kwijt te spelen wordt I'_1gecompileerd in I'_1 door I_2 te specialiseren in I'_1 :

$$I'_1(\text{exp},\text{env}_1,\text{env}_2) = \text{PE}(I_2(\text{I}'_1(\text{exp},\text{env}_1),\text{env}_2,\text{env}_3))$$

Hier zijn exp, env_1 en env_2 zijn onbekende parameters, terwijl env3 een gekende parameter is dewelke de body bijhoudt van level 2 functies wanneer de compilatie plaats heeft. We gebruiken de waarde van env_3 als een gekende waarde omdat we anders zonder dat nogal moeilijk kunnen partieel I_2 evalueren.

Omdat we de waarden van env_3 nodig hebben als een gekende waarde zullen veranderingen die aangebracht worden aan env_3 na de partiele evaluatie geen effect hebben op het gedrag van I'_1. Gecompileerde programma's zullen veranderingen aan de metalevel interpreter zelf negeren.

Deze partiele evaluator kan ook gebruikt worden om gebruikersprogramma's te compileren. Om het programma P te compileren, specialiseren we I_1 op P als volgt :

$$P(\text{arg},\text{env}_1) = \text{PE}(I_1(\text{P}(\text{arg}),\text{env}_1,\ \text{env}_2))$$

waar exp en env_1 onbekende parameters zijn, terwijl env_2 de waarde bijhoudt van env_2 tijdens partiele evaluatie tijd.

9 Conclusie

In dit eerste deel hebben we een overzicht gegeven van een aantal technologien.

Als eerste hebben we de Reflectieve Virtuele Machine bekeken. Deze virtuele machine laat ons toe om programma's en de interpretatie van programma's at runtime te manipuleren. De bedoeling van deze machine is om de computational state van een programma te kunnen vatten, deze te migreren naar een andere virtuele machine, en daar de computation verder te zetten. Reflectie wordt mogelijk gemaakt door een oneindige toren van interpreters, maar het is juist deze oneindige toren die Reflectieve Virtuele Machines juist zo traag maakt.

Als tweede technologie hebben we Pico/Borg onder de loep genomen. Borg is een mobiel multi-agent platvorm. Het is een reflectieve virtuele machine en het zal de basis vormen van deze thesis. We zullen in Borg onze bevindingen testen.

In het tweede deel hebben we dan een aantal RVM's onder de loep genomen. We zijn begonnen met RbCl. Deze reflectieve taal is een kernel-less systeem. Dit wil zeggen dat er een mini-kernel bestaat die boven het OS en de hardware staat. De programmeertaal is in een metacirculaire vorm. Dit laat toe dat de volledige programmeertaal aanpasbaar is, tot zelfs de garbage collector toe. RbCl is een zeer aanpasbaar systeem, maar het heeft een prijs moeten betalen, en dat is performantie.

Als tweede systeem hebben we dan de Virtuele Virtuele Machine (VVM) gezien. Dit systeem laat ook weer toe om zeer aanpasbaar te zijn at runtime, maar heeft hierbij ook weer met performantie moeten betalen. Dit systeem maakt in tegenstelling tot RbCl wel gebruik van een kernel, dewelke onveranderbaar is.

Vervolgens was NitrO aan de beurt. Ook dit is een at runtime aanpasbare programmeertaal. Maar al gauw moesten we tot dezelfde constatatie komen als de andere reflectieve systemen. Zeer aanpasbaar maar traag doordat een twee-level interpreter toren gebruikt wordt.

En als laatste hebben we Black gezien. Hier heeft men restricities gezet op de aanpasbaarheid van het systeem om performantie winst te bekomen en waarbij men gebruik maakt van partiele evaluatie.

We kunnen concluderen dat Reflectieve Virtuele Machines "traag" zijn omdat zij een oneindige toren van interpreters bevatten en als we met interpreters werken maar toch snelheid willen verkrijgen, we hiervoor delen van de aanpasbaarheid moeten opofferen.

We zullen dit proberen oplossen door JIT-compilatie te introduceren, en de interpreter van de geheel aanpasbare systemen (zoals RbCl) met deze

jit-compiler te vervangen.

Deel III
Reflectie door middel van Jit-compilatie

In het vorige deel hebben we een aantal interessante technologien gezien. Namelijk het concept van de reflectieve virtuele machine die toelaat om programma's en de interpretatie van programma's te manipuleren. Het nadeel van deze technologie is dat deze voorlopig nog aan de trage kant is. Een andere technologie is het concept Jit-compilatie, dat voor een performantie winst zorgt bij het uitvoeren van Java programma's. En uiteindelijk hebben we ook nog RbCl gezien. Een kernel-less systeem dat een programmeertaal inlaadt. Deze laatste technologie heeft als voordeel dat de volledige programmeertaal aanpasbaar is, en deze aanpassingen op het laagste niveau gebeuren.

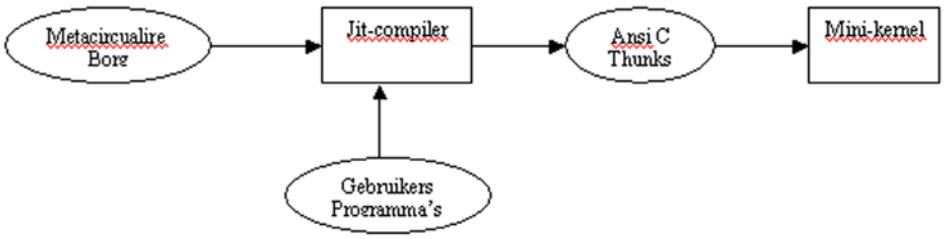
Deze technieken kunnen we nu tesamen brengen en komen tot de definitie van een nieuwe Reflectieve Virtuele Machine.

10 De RVM met Jit-compilatie

> The statement below is true.
>
> The statement above is false. – **Anonymous**

10.1 Inleiding

In hoofdstuk 2 hebben we al kennis gemaakt met de Reflectieve Virtuele Machine en in hoofdstuk 5 dan met RbCL, een kernel-less reflectieve programmeertaal. Het is nu de bedoeling om de definitie van de Reflectieve Virtuele Machine uit hoofdstuk 2, uit te breiden met het begrip kernel-less systeem. Dit wil dus zeggen dat de Refectieve Virtuele Machine zal bestaan uit een mini-kernel die een metacirculaire representatie zal inladen. Maar metacirculaire interpreters hebben nogal de neiging om aan de trage kant te zijn omdat er een extra layer van interpretatie wordt toegevoegd. Daarbij komt ook nog eens dat reflectieve talen opgebouwd worden aan de hand van een oneindige toren van interpreters (cfr. 2.5). Maar wij zijn ervan overtuigd dat Jit-compilatie deze oneindige toren van interpreters kan samenbrengen (compileren) tot 1 interpreter.



Figuur 26: Design van Reflectieve Virtuele Machine

10.2 Design

Het design van deze reflectieve virtuele machine ziet er als volgt uit. We creeeren een kernel-less systeem. Dit wil zeggen dat we een minimale kernel voorzien die enkel instaat voor de connectie met het onderliggende OS en de onderliggende hardware. De volledige taal zit in een metacirculaire voorstelling. Het is nu de bedoeling om deze metacirculaire voorstelling bij deze mini-kernel toe te voegen zonder dat er nood is aan een interpreter die deze metacirculaire voorstelling moet interpreteren (en zo zorgt voor een extra layer). Dit doen we door absorbtie[7] en reificatie[8] door middel van Jit-compilatie. Hoe dit in zijn werk gaat leest u verder in punt 10.6. Anders gezegd gaan we onze metacirculaire representatie jit-compileren naar Ansi C thunks. (figuur 26)

Ook gebruikersprogramma's worden door onze Jit-compiler bij de mini-kernel ingeladen evenals worden composities van reeds bestaande functies en natives gecreeerd. We kunnen deze gebruikersprogramma's namelijk zien als een uitbreiding van het Borg systeem.

10.3 Problemen

In deze subsectie gaan we ons eerst buigen over de mogelijke problemen die we kunnen tegenkomen. Tot op heden hebben we nog geen literatuur gevonden die dezelfde kijk heeft op reflectieve virtuele machines als de onze, nl. gebruik maken van Jit-compilatie om een performantere reflectieve virtuele machine te bekomen, en dit door de oneindige toren van interpreters te jit-compileren tot 1 interpreter, 1 layer.

Vooreerst wordt reflectie ondersteund door interpretatie. Reflectie laat toe om veranderingen aan te brengen aan de programmeertaal. Deze veran-

[7]Absorbtie : vanuit ANSI C, pico gebruiken.
[8]Reificatie : vanuit Pico, ANSI C gebruiken

deringen moeten toegepast worden in de uitvoering van programma's en dit kan enkel als het programma geinterpreteerd en uitgevoerd wordt in de veranderde programmeertaal. Eenmaal een programma gecompileerd is, staat zijn uitvoering vast, en kan daar niets meer aan veranderd worden. Veronderstel een programma P geschreven in Scheme. Als P geinterpreteerd wordt door een Scheme interpreter I zoals in I(P,env) dan kunnen we delen van P wijzigen. Bijvoorbeeld als we de functie f in P willen veranderen in g is 'set! f g' het enige dat we moeten doen. Na dit is de waarde van f in de environment veranderd naar g. Omdat de environment elke keer bij een functieoproep wordt nagekeken wordt g uitgevoerd wanneer f wordt opgeroepen. We kunnen ook meer gesofistikeerde veranderingen aanbrengen door gebruik te maken van reflectieve faciliteiten. Veronderstel dat we de volgorde van evaluatie van argumenten willen veranderen. Dit wordt meestal beschreven in eval-list (of evlist) in I. Dus als we eval-list veranderen door een nieuwe my-eval-list gedefinieerd door de gebruiker, gebruik makende van de reflectieve faciliteiten dan zal de evaluatie volgorde veranderd zijn door diegene gedefinieerd door my-eval-list. Omdat P geinterpreteerd wordt door I kunnen we het gedrag van P aanpassen als we veranderingen mogen aanbrengen aan I. Dit kan omdat de interpreter I alle informatie heeft hoe P moet geinterpreteerd worden. Wat als we P nu compileren in machine code *P* en deze direct uitvoeren. Dan zullen alle wijzigingen die hierboven beschreven zijn niet meer mogelijk zijn omdat *P* direct uitgevoerd wordt. Zelfs al zouden we de uitvoering stopzetten en f veranderen in g, dan nog zou de uitvoering niet veranderen omdat *P* gecompileerd is en geen opzoeking in het environment meer doet. De herdefinitie van eval-list zou ook het gedrag van *P* niet meer beinvloeden omdat *P* niet meer geinterpreteerd wordt door I. Compilatie heeft dus een hoge efficientie omdat het er van uit gaat dat het programma niet meer veranderd wordt. Als we dus de veranderingen die hierboven beschreven staan willen toelaten moet P dus expliciet geinterpreteerd worden door gebruik te maken van I. Willen we toch de veranderingen ook op dit gecompileerd programma tot stand brengen, dan moeten we dit programma gaan hercompileren. Interpretatie is dus het kernwoord.

Jit compilatie is zoals we gezien hebben in hoofdstuk 4, de samenstelling van interpretatie en compilatie. Er zit dus een deeltje interpretatie in Jitcompilatie, waardoor Jit een mogelijke kanshebber kan zijn om performantere reflectieve virtuele machines te creeeren. Rest ons enkel nog de vraag of het compilatie gedeelte te verzoenen is met onze reflectieve virtuele machine. Het antwoord hierop is 'ja', als we gebruik maken van een versie / dependency systeem. Dit bespreken we in punt 10.4.

We hebben zojuist gesteld dat een JIT-compiler kan gebruikt worden voor een RVM. Een ander (kleiner) probleem dat hiermee te voorschijn komt, is

dat sommige functies beter zouden geinterpreteerd worden dan gecompileerd, omdat deze maar 1 keer worden gebruikt en compilatie dan te duur wordt. Voor functies die maar 1 keer opgeroepen worden zou interpretatie beter zijn. Hier komt SMART-Jit ons ter hulp. Meer hierover in punt 10.5.

10.4 Versie- / Dependency systeem

Zoals in punt 10.3 aangehaald is compilatie niet te verzoenen met reflectie. Om iets te kunnen wijzigen, zowel aan het programma zelf als aan de programmeertaal, moet er geinterpreteerd worden. Als we compileren moet bij elke wijziging opnieuw gehercompileerd worden. Daar we nu met Jit-compilatie werken moet niet het volledige programma ge-hercompileerd worden, maar enkel het deel waarbij er iets veranderd is. Tot op heden is het nog niet duidelijk welke blokken per keer ge-jitcompileerd worden (in java is dat per file of anders gezegd per klasse), in Borg denken we per functie te jit-compileren. Dus als 1 functie geherdefinieerd wordt, of 1 native. Dan moet enkel die native gehercompileerd worden.

Het is goed mogelijk dat de nieuwe versie afhankelijk is van de oude versie, Bijvoorbeeld de 'plus' native, waarbij de nieuwe 'plus' beroep dat op de oude 'plus' :

```
myplus(a, b):
  {
    display(a," plus ", b, eoln);
    a+b
  }
+:myplus
```

Daar de 'oude' plus nog steeds gebruikt wordt moet deze blijven bestaan. De oude thunks moeten blijven bestaan, en er moeten nieuwe thunks gecreeerd worden. Om geen dubbele continuatie namen te krijgen hebben we hier nood aan een versiesysteem die namen aan continuaties toekent.

Als we daarna de oude plus aanpassen, moet deze ook aangepast worden in de nieuwe versie. We doen namelijk niet meer aan interpretatie en zoals in punt 10.3 reeds uitgelegd zal een gecompileerd programma geen opzoekingen meer doen in het environment, noch de interpreter wordt gebruikt. Dus ook de nieuwe plus moet gehercompileerd worden. Deze afhankelijkheden moeten ook bijgehouden worden. Daarvoor zal het versiesysteem / dependecy systeem ook moeten instaan.

De oplossing om in de continuatie van de nieuwe plus, de continuatie van de oude plus op de EXP-stack te steken zou neerkomen op het implementeren van een directe implementatie van een oneindige toren van interpreters, en is nog steeds trager dan 1 enkele interpreter. Daarbij maken we ons sterk dat we met een JIT-compiler werken, en dat bij de verandering van de oude plus bijvoorbeeld, niet direct de nieuwe plus ook gecompileerd hoeft te worden. Enkel bij de oproep van de nieuwe plus zal ook deze gehercompileerd worden. Hier zou nog verder onderzoek over kunnen gedaan worden, nl. hoe groot moet een thunk zijn om optimale performantie te hebben en bij hercompilatie een minimum aan compilatie te hebben. Een voorbeeld. Stel we hebben de '<' operator, de 'or'-operator en de '='-operator. We kunnen deze operatoren samenstellen tot de '<='-operator. Stel dat we nu enkel de '<' operator zouden veranderen. Als we de '<='-operator in 1 thunk gecompileerd hebben, moeten we inderdaad de volledige thunk hercompileren. Als we daarentegen alles naar meerdere kleinere thunks hebben doorgecompileerd, hoeven niet alle thunks gehercompileerd te worden, enkel deze maar die tot de '<'-operator behoren moeten opnieuw gecompileerd worden.

Het hoofdidee bij het splitsen in thunks en jit-compilatie is performantie, en dit vooral bij samenstellingen. Laat ons nogmaals bovenstaand voorbeeld nemen. Als we een '<' uitvoeren wordt eerst gecontroleerd of de input wel getallen zijn. Hetzelfde bij de '='. We zitten hier dus met dubbele code, 2 maal dat gecontroleerd wordt of we wel de juiste input hebben. Het is nu de bedoeling om de actie te strippen van zijn controle-mechanisme, en bij de compositie enkel nog met de relevante controle over te blijven, en dit maar eenmalig. Bijvoorbeeld als de '='-operator ook kan gebruikt worden voor strings, moet er gecontroleerd worden of de intput getallen zijn of strings, eenmaal bij de compositie in de '<='-operator, mag de controlle op strings weg vallen.

Het versie-systeem / Dependency systeem is nog een moeilijke brok.

10.5 SMART-Jit

Zoals reeds aangehaald hierboven is het mogelijk dat in sommige gevallen interpretatie performanter kan zijn dan Jit-compilatie om de eenvouige reden dat de functie maar 1 keer opgeroepen wordt. Ook bij functionele parameters is het meer dan waarschijnlijk dat telkens een andere functie als parameter wordt meegegeven en zo een andere betekenis aan een functie geeft. Als we zo een functie zouden (jit)-compileren is dit maar voor eenmalig gebruik en dus veel te duur.

We moeten dus een systeem hebben dat ofwel interpreteert ofwel compileert, en dit zo performant mogelijk. En hier komt de SMART-Jit om de

hoek kijken. SMART-Jit is een systeem dat alles minimaal 1 keer interpreteert. De functies die meerdere malen opgeroepen worden, worden dan na een tijdje ge-jit-compileerd. Zo vermijdt men dure onnodige compilatie. Een neveneffect hierbij is het "heating up" effect omdat in de eerste cycle er geen snelheidwinst is tegenover interpretatie. Eenmaal functies meerdere malen gebruikt zijn, en gecompileerd worden begint het effect van performantie winst te spelen.

Naar de toekomst toe heeft SMART-Jit nog een interessante toekomst, want de huidige computers kunnen reeds meer dan 1 miljard instructies per seconde uitvoeren, maar de 'bottleneck' ligt eerder bij het lezen van disk (of soms zelf van geheugen). Een compacte (high level code) representatie zal sneller van disk gelezen worden dan een lange (low level machine code), maar performante, representatie van de code. De gebruiker let niet zozeer op het feit of uw CPU nu idle is en wacht voor IO of voor geheugen toegang, of dat de CPU dure berekeningen zit te maken. Het enige wat telt is de tijd die er over gedaan wordt tussen de eigenlijke vraag (muisklik of toetsaanslag) en het verschijnen van het resultaat op het scherm. En SMART-Jit zal compacte broncode inlezen en deze op een performante en snelle manier uitvoeren i.p.v. lang te wachten op het uitlezen en inladen van de low-level code die wel snel kan uitgevoerd worden.

Zoals ook al aangegeven in het punt 10.4 is het mogelijk dat reeds gecompileerde stukken code moeten gehercompileerd worden. Dit omdat een deel van de code waarvan het afhankelijk is, gewijzigd is. Indien dit stuk code reeds gecompileerd was, mogen we er niet meer vanuit gaan dat deze code reeds meermaals gebruikt werd, en moeten we dus opnieuw de eerste maal interpreteren. Dit vooral omdat het kan zijn dat we in een debugfase van het programmeren zijn, en dus meerdere malen aanpassingen zullen doen en snel willen weten of de nieuwe code in orde is. Als we moeten wachten tot het geheel gecompileerd is, is dit te duur. Bij een tweede test zonder verandering kunnen we overgaan tot compilatie.

10.6 Linguistic Symbiosis

Tot hiertoe hebben we gekeken naar de simpele/theoretische kant van de zaak. In deze sectie gaan we dieper in op de praktische kant hoe we de metacirculaire voorstelling in de minikernel gaan krijgen. De technologie die hier achter zit noemt linguistic symbiosis en hebben we ook al eventjes besproken in 5.2.1.

We beginnen met een kernel-less systeem en die moet de metacirculaire voorstelling inladen. Laat ons als voorbeeld de plus-operator nemen. In de metacirculaire voorstelling moeten we deze plus voorstellen in Borg, maar er

is niets voorhanden om die plus voor te stellen, want er is nog niets in Borg. De enige oplossing is dat we de plus koppelen aan de C plus operator. Deze manier van programmeren noemt men ook nog wel expliciet programmeren, omdat we de Pico-plus expliciet gaan koppelen aan de C-plus. We gebruiken in onze Pico omgeving nu een C-operator. Door middel van Jit-compilatie vertalen we nu deze representatie naar C-code en laden die bij in de minikernel. Vanaf dat moment kunnen we de '+' gebruiken in Pico, en deze hangt af van de '+' van C. Dat we nu gebruik kunnen maken van '+' noemt men reificatie. Want eigenlijk maken we gebruik van de C-plus maar dan binnen Pico.

Het omgekeerde moet ook mogelijk zijn, vanuit Ansi C, Pico code oproepen. Dit noemt men dan absorbtie. Maar dit is heel wat moeilijker dan de reificatie. De reden hiervoor ligt dat midden in de uitvoering van een thunk mogelijk Pico code moet uitgevoerd worden. Nadat we deze Pico code hebben uitgevoerd moeten we verder gaan waar we gestopt zijn, nl. middenin de continuatie. We kunnen hiervoor threads of CPS (Continuation Passing Style) of nog andere mogelijke oplossingen gebruiken. Wij opteren voor CPS want CPS gaat zijn toekomst meegeven aan een thunk. Zo weet de thunk, nadat deze is uitgevoerd, wat er moet gebeuren, want dat is wat hij namelijk als parameter heeft meegekregen (zijn toekomst). Dus in plaats van de toekomst (de thunks die later moeten uitgevoerd worden) op een stack te plaatsen zoals bij Borg, worden deze thunks meegegeven met de huidig uitgevoerde thunk. In het geval bij ons, gaan we hetgeen na de Pico-code komt (binnen dezelfde thunk) meegeven (CPS) aan de Pico code, zodat deze weet wat er na het uitvoeren van de Pico code verder moet gebeuren.

De technologie van de absorbtie is bijvoorbeeld nodig bij het uitvoeren van een if-statement. We kunnen op het ogenblik van compilatie nog niet weten welke tak (de then-tak of de else-tak) zal uitgevoerd worden, omdat de conditie nog niet geevalueerd is. Pas bij de uitvoering van de gecompileerde code zal duidelijk worden welke tak men zal gebruiken. Beide takken worden nog niet op voorhand gecompileerd, enkel wanneer geweten is welke tak gebruikt zal worden, wordt deze gecompileerd. Dit wil dus zeggen dat in de gegenereerde C-code van de if-statement, zich nog steeds Pico code zal bevinden, dewelke dan geevalueerd moet worden.

10.7 Wanneer en wat Jit-compilen?

Nu moet men nog beslissen wanneer en wat men gaat Jit-compileren. Als men beslist om het volledige programma te Jit-compilen, dan is men eigenlijk bezig met compilatie juist voor de uitvoering (en dat is niet snel in gebruik). In Java compileert men klasse per klasse. Dit is mogelijk omdat alles een

klasse is in Java, en elke klasse zit in een aparte file. Dus wordt er per keer een volledige file ge-jit-compileerd. In java is het dus zeer duidelijk wat gecompileerd wordt. In Borg splitsen we geen klassen in files op. Dus hier ligt het iets moeilijker om te beslissen wat men nu per deel gaat compileren. Wat voor de hand lijkt te liggen is te jit-compileren per functie. Tot op heden hebben we hier nog geen oplossing voor gezocht. We hebben elke keer zelf beslist wat en wanneer ge-jit-compileerd moest worden.

Ook moet er onderzoek geleverd worden naar de optimale grootte van een thunk. Aan de ene kant hebben we grote thunks die snel zijn in uitvoering (er moet niet telkens met de stack gewerkt worden) maar die niet optimaal zijn voor garbage-collection, user-interaction en hercompilatie. Het is tussen de uitvoering van thunks (praktisch gezegd bij aanvang van het uitvoeren van een thunk) dat garbage-collection kan plaats hebben. Dus grote thunks laten de garbage-collector langer wachten om op te kuisen. Ook user-interaction gebeurt tussen het uitvoeren van 2 thunks. Aan de andere kant hebben we zeer kleine thunks die optimaal zijn voor de compilatie, garbage-collection en userinteraction, maar die het systeem tergend langzaam maken omdat er constant thunks en data van de stack moet gehaald worden of er terug opgestoken worden. Zoeken naar de ideale grootte van de thunk valt buiten de scoop van deze thesis.

10.8 Dynamisch linken[21]

In de vorige subsecties hebben we ons bezig gehouden met jit-compilatie. Maar nu eenmaal C-code gegenereerd werd, wat moet er dan gebeuren. Deze C-code moet op zijn beurt omgevormd worden naar machine-taal (nog even ter herinnering dat we naar Ansi C code compileren voor het platvorm onafhankelijk karakter van deze taal). Borg (de mini-kernel) draait ondertussen al, en dus moeten deze nieuwe functies dynamisch gekoppeld worden aan het geheel. Dit gebeurd door middel van dynamische libraries. Deze kunnen tijdens de uitvoering van het programma ingeladen worden bij de rest van het programma.

10.9 Werking

Wat is nu het volledige idee? Wel, een programmeur start Borg. Dit wil zeggen dat hij de mini-kernel opstart. Daar het een mini-kernel betreft zal deze snel ingeladen worden. De programmeur begint met het schrijven van een functie. Eenmaal dat gedaan wil hij deze functie natuurlijk eens testen. Op dit moment is het de eerste maal dat hij deze functie nodig heeft, dus deze functie wordt geinterpreteerd (omdat het de eerste maal is, SMARTJIT).

Omdat er ook natives in deze functie zitten, zullen ook deze natives ingeladen worden en geinterpreteerd worden. Er zit een kleine bug in de functie en onze programmeur wijzigt iets aan zijn functie en roept opnieuw de functie op. Opnieuw is het de eerste maal dat de 'nieuwe' functie wordt opgeroepen, dus opnieuw wordt de functie geinterpreteerd. Maar op dit moment worden de natives die gebruikt worden, en niet veranderd zijn, voor een 2de maal opgeroepen en worden ge-jit-compileerd en als dynamische library bij het systeem ingeladen. Vanaf dit moment zullen deze natives direct uitgevoerd worden in plaats van geinterpreteerd te worden. Nu is de functie van onze programmeur in orde en hij test zijn functie nogmaals met andere waarden. Op dit moment is het ook de 2de maal dat deze functie wordt opgeroepen en wordt deze ook gecompileerd en geoptimiseerd. Een optimisatie betekent dat de natives worden ge-inlined en dus in de thunk van de functie komen te zitten, i.p.v. een native-thunk op stack te plaatsen, en deze reeds gecompileerde native-thunk te gebruiken (cfr. 10.4).

11 Conclusie

> "A conclusion is symply the place where you got tired of thinking."
> – **Anonymous**

We hebben gezien dat in de hedendaagse reflectieve systemen er een balans moet gezocht worden tussen snelheid en aanpasbaarheid. Systemen die at runtime alles kunnen veranderen zijn zeer traag en als we performantie winst willen verkrijgen, dan moeten we daarvoor (gedeeltelijk) reflectie voor opofferen.

In deel 2 hebben we daaromtrent een overzicht gegeven. We hebben de hedendaagse reflectieve virtuele machines van dichterbij bekeken en gezien dat de snelle machines niet geheel aanpasbaar waren en de aanpasbare machines dan weer heel traag waren.

Uit deel 1 hebben we onthouden dat interpretatie het sleutelwoord is voor reflectieve virtuele machines. Daarenboven hebben we ook het concept jit-compilatie gezien dat de voordelen van compilatie en interpretatie probeert te combineren.

In deel 3 hebben we dan onderzocht of de compilatie factor uit jit-compilatie (in samenwerking met de interpretatie factor) te verzoenen was met reflectie. En het antwoord is dat Jit-compilatie inderdaad te verzoenen is met reflectie.

Wij hebben nu gezien dat om performantie winst te bekomen we daar geen reflectieve faciliteiten voor moeten opofferen. Jit-compilatie in combinatie met een versie- / dependency systeem kan geimplementeerd worden en zo voor performantie winst zorgen. Direct uitvoerbare systemen zijn per definitie snel en door ze modulair te maken is het mogelijk enkel de veranderde modules opnieuw te jit-compileren en zo voor een aanvaardbare snelheid zorgen en toch geheel aanpasbaar zijn.

12 Future work

Elke keer Borg opgestart wordt, moeten we opnieuw de frequent gebruikte functionaliteiten gaan jit-compileren. Deze stap is noodzakelijk daar men met een meta-circulaire representatie bezig is. Misschien is het mogelijk om bij het afsluiten van de virtuele machine een memory-image te nemen, en deze te bewaren, zodat bij het opstarten van Borg de volgende keer, enkel deze image in het geheugen moet geladen worden, i.p.v. te jit-compileren. Enkel bij veranderingen hoeft men onze Jit-compiler dan nog maar te gebruiken.

Het zoeken van een optimale grootte van een thunk is ook een gebied dat niet tot deze thesis behoort maar wat wel belangrijk is voor deze thesis. Hoe groter een thunk, hoe sneller dat hij kan uitgevoerd worden (er moet niet constant data op en van de stack gehaald worden), maar hoe onstabieler het systeem wordt. Middenin een thunk kan er namelijk geen garbage-collection gedaan worden. Dit gebeurd telkens bij aanvang van een thunk. Ook user interaction wordt tussen 2 thunks afgehandeld. Maar bij zeer kleine thunks wordt het systeem uitermate traag. Het probleem stelt zich vooral bij composities, waarin 2 (of meer) bestaande thunks tesamen worden gebruikt in een nieuwe functie/native. Wat is dan de optimale grootte zodat we een snel systeem hebben, maar toch ook weer niet alle thunks moeten hercompileren. Hieromtrent moet ook nog onderzoek gedaan worden.

Security is ook niet in deze thesis behandelt. Het was in deze thesis vooral de bedoeling om een Reflectieve Virtuele Machine met Jit-compilatie te beschrijven en te onderzoeken of JIT-compilatie te verzoenen is met reflectie. In onze reflectieve taal is alles aanpasbaar. Het beveiligen van deze systemen behoorde niet tot de scoop van deze thesis.

En nu we toch bezig zijn met een metacirculaire Borg kan er ook onderzocht worden tot hoever de Jit-compiler in Borg kan geschreven worden. Als we naar het voorbeeld van RbCl (punt 5) kijken waarbij ook een kernelless systeem gebruikt werd, maar zonder jit-compiler, kunnen we vanuit dit startpunt eerst onze Borg-Jit-compiler interpreteren en vanaf dan alles jit-compileren zodat we zelfs d.m.v. reflectie de Jit-compiler volledig kunnen aanpassen en optimiseren.

Referenties

[1] Karsten Verelst, Werner Van Belle, Theo D'Hondt: The Reflective Virtual Machine, Vrije Universiteit Brussel, 2001

[2] Borg, Vrije Universiteit Brussel, http://borg.rave.org

[3] Pico, Vrije Universiteit Brussel, http://pico.vub.ac.be

[4] Object oriented languages: Mobile Agents, http://www.cetus-links.org/oo_mobile_agents.html

[5] Werner Van Belle, Karsten Verelst, Theo D'Hondt: The Cborg Mobile Multi-Agent System, Vrije Universiteit Brussel

[6] Werner Van Belle, Theo D'Hondt: Agent Mobility and Reification of Computational State, Vrije Universiteit Brussel, 20 maart 2000

[7] John Aycock: A brief History of Just-In-Time, University of Calgary

[8] Francisco Ortin Soler, Juan Manuel Cueva Lovelle: Building a completely adaptable Reflective System, Campus Llamaquique Oviedo (Spain),

[9] Gregoer Kiczales, J. Des Rivieres, D.G. Bobrow: The Art of Metaobject Protocol, MIT Press, 1992

[10] Bertil Folliot, Ian Piumarta, Fabio Riccardi: A Dynamically Configurable, Multi-Language Execution Platform, INRIA Roquencourt (France)

[11] Chuck McManis: Just In Time Compilation, http://www.javacats.com/us/articles/chuckmcmanis091696.html

[12] Yuuji Ichisugi, Satoshi Matsuoka, Akinori Yonezawa: RbCl : A Reflective Object-Oriented Concurrent Language without a Run-time Kernel, University of Tokyo

[13] Java JIT compiler overview, Sun Microsystems, http://www.sun.com/solaris/jit

[14] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu and Toshio Nakatani: Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler, IBM Tokyo Research Laboratory (Japan)

[15] Kenichi Asai, Satoshi Matsuoka, Akinori Yonezawa: Duplication and Partial Evaluation - For a better understanding of reflective languages, october 1993, University of Tokyo

[16] Kenichi Asai, Satoshi Matsuoka, Akinori Yonezawa: Roles of a partial Evaluator for the Reflective Language Black , 1994, University of Tokyo

[17] Schemers, http://www.schemers.org

[18] Urs Holze: A brief History of Just-In-Time , UCSB, http://www.cd.ucsb.edu/~urs

[19] Michael Golm: Design and Implementation of a Meta Architecture for Java, Friedrich-Alexander-Universitat(Germany), 1997, Erlangen

[20] Michael P. Plezbert and Ron K. Cytron: Does "Just in Time" = "Better Late than Never"?, Washington University Box 1045, Departement of Computer Science, St. Louis, MO 63130 USA

[21] Tom: http://www.gerbil.org/tom/archives/tom/tom.99xx/msg00208.html.

[22] http://www-106.ibm.com/developerworks/linux/library/l-dll.html?dwzone=linux, dynamische libraries in Linux

[23] Glen McCluskey: Using Java Reflection, 1998, http://developer.java.sun.com/technical/Articles/ALT/Reflection/

[24] Justin Kim, Introduction to java.lang.reflect package, http://www.cs.auckland.ac.nz/sdk/journal1/reflect/

[25] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, T. Nakatani: Overview of the IBM Java Just-In-Time Compiler, IBM Systems Journal, 2000