Inhoudsopgave

Inleiding **6**

1 IEEE 754, de floating point standaard **8**
1.1 Los van talen 9
1.2 Precisie 9
1.2.1 Fortran 90/95 9
1.2.2 C++ 11
1.2.3 Haskell 11
1.3 Representatie 11
1.3.1 Fortran 90/95 11
1.3.2 C++ 13
1.3.3 Haskell 13
1.4 Excepties 13
1.4.1 Gebruik 14
1.4.2 Fortran 90/95 15
1.4.3 C99/C++ 16
1.4.4 Haskell 17
1.5 Conclusie 17

2 Objectgericht ontwerp en numerieke toepassingen **19**
2.1 Inleiding 19
2.2 Architectuur 20
2.2.1 Inlining 20
2.2.2 Aliasing 21
2.3 Klassen 22
2.3.1 Methodes 22
2.3.2 Encapsulatie 22
2.3.3 Overerving 22
2.3.4 Excepties 23
2.3.5 Tijdelijke objecten 23
2.4 Polymorfisme 26
2.4.1 Templates 26
2.4.2 Expression templates 28
2.5 Conclusie 34
2.6 Zin en onzin van technisch wetenschappelijk ontwerp in Java 35
2.6.1 Snelheid van de code 36

2.6.2 Eenvoud in programmeren 37
2.6.3 Eventuele oplossingen 39
2.6.4 Conclusie 40

3 Functionele talen **41**
3.1 Mogelijkheden 41
3.1.1 Analogie met de wiskunde 42
3.1.2 Functies 42
3.1.3 Currying en hogere orde functies 43
3.1.4 Pattern matching en guards 44
3.1.5 Lazy evaluation 44
3.1.6 Zelfgedefinieerde datatypes 44
3.1.7 Overloading en parametrisch polymorfisme 47
3.1.8 Overerving 48
3.1.9 Operatoren 48
3.2 Typesysteem 48
3.2.1 Groepentheorie 49
3.2.2 Toepassingen 49
3.2.3 Natuurlijke getallen 52
3.2.4 Type inference 53
3.3 Lazy evaluation en machtreeksen 53
3.3.1 Machtreeksen 53
3.4 Problemen 58
3.5 Conclusie 60

4 Vergelijking C++ en Haskell **61**
4.1 Implementatie van problemen 61
4.1.1 Diffusieprobleem 61
4.1.2 Lattice Rules 65
4.2 Functies 71
4.2.1 Typespecificatie 71
4.2.2 Lambdafuncties en de interactieve commandolijn 71
4.2.3 Structuur 71
4.2.4 Hoofding 71
4.2.5 Doorgeven 72
4.2.6 Pure functies versus niet-pure functies 72
4.3 Typesysteem 73
4.3.1 Beperkingen op types 73
4.3.2 Wiskundig denken over types 73
4.3.3 Typeconversie 74
4.3.4 Overloading 74
4.3.5 Pointers, geheugenbeheer en ongeïnitialiseerde waarden 76
4.4 Modellering 77
4.4.1 STL versus datatypes 77
4.4.2 Datatypes versus klassen 77
4.4.3 Lazy evaluation 78
4.5 Debuggen 78

4.5.1 Informatie tijdens de uitvoering tonen 78
4.5.2 Tellen 79
4.6 Conclusie 81

5 Besluit **82**

A Code **84**
A.1 Diffusieprobleem 84
A.1.1 C++ versie 84
A.1.2 Haskell versie 91
A.2 Lattice rules 96
A.2.1 C++ versie 96
A.2.2 Haskell versie 109

Lijst van figuren

2.1 Callback inlining . . . 21
2.2 Tijdelijke objecten bij toekenningen vermijden . . . 24
2.3 Initialisatie van data members. . . . 25
2.4 Optimalisatie van return values . . . 26
2.5 De *Barton and Nackman Trick* . . . 27
2.6 Fortran **WHERE** expression template code . . . 29
2.7 Fortran **WHERE** expression template code . . . 31
2.8 Expression templates . . . 32
2.9 Uitvoeringstijd van expression template code . . . 33
2.10 Verschil tussen code met operaties op complexe getallen zonder en met operator overloading. . . . 38

3.1 Conjugate gradients in Haskell . . . 43
3.2 Pattern matching en guards . . . 45
3.3 Zelfgedefinieerde datatypes . . . 46
3.4 Overloading . . . 47
3.5 Groepentheorie . . . 50
3.6 Toepassingen . . . 51
3.7 Operaties op machtreeksen . . . 54
3.8 Toepassing van functies op machtreeksen . . . 55
3.9 Andere bewerkingen op machtreeksen . . . 56

4.1 Oplossing van de warmtevergelijking voor koper . . . 66
4.2 Oplossing van de warmtevergelijking voor aluminiumoxyde . . . 67
4.3 Code van **itereerTotEinde** in Haskell om na elke iteratiestap het grootste verschil in het domein te tonen. . . . 72
4.4 Overloading en typesysteem . . . 75
4.5 Debuggen in Haskell . . . 80

A.1 Klassendiagram van het diffusieprobleem . . . 85
A.2 Klassendiagram van lattice rules . . . 97

Lijst van tabellen

1.1 Overzicht van gelijkaardige functies in Fortran, C++ en Haskell 12

A.1 Overzicht van de methodes in de C++ klasse **diffusion2d** 86
A.2 Overzicht van de functies in de Haskell versie van het diffusieprobleem 92
A.3 Overzicht van de methodes in de C++ klassen voor lattice rules. 98
A.4 Overzicht van de Haskell versie van het latticeruleprobleem 110

Inleiding

Computers worden sinds hun ontstaan voor technisch-wetenschappelijke of numerieke toepassingen gebruikt. De term computer werd in oorsprong gebruikt voor iemand, die onder de leiding van een wiskundige berekeningen uitvoert (Wikipedia [2006]). De drijvende kracht achter de ontwikkeling van computers was juist het oplossen van numerieke problemen, zoals het ontcijferen van gecodeerde berichten tijdens de tweede wereldoorlog. Het is dan ook niet toevallig dat de eerste algemeen gebruikte hogere programmeertaal in dit domein ontstond. In 1954 verschijnt Fortran op het toneel, en deze taal wordt tot op vandaag nog actief gebruikt voor numeriek ontwerp.

In dit licht gezien is het op zijn minst vreemd te noemen, dat niemand ooit de moeite schijnt genomen te hebben om de programmeertalen die gebruikt worden voor numeriek ontwerp, naast elkaar te plaatsen en te kijken op welke vlakken een taal zich wel of niet leent tot numeriek ontwerp. In tegendeel, veel mensen schijnen klakkeloos de taal over te nemen van de mensen die hen voorgegaan zijn, zonder zich ook maar de vraag te stellen of de taal eigenlijk wel geschikt is om het probleem in kwestie te lijf te gaan.

Deze eindverhandeling zet een eerste stap om deze leemte aan te vullen. Alle programmeertalen bespreken, die voor numeriek ontwerp gebruikt worden, zou ons niet toelaten om een vergelijkende studie te maken in het tijdsbestek dat ons gegeven is. Je moet tenminste voldoende tijd hebben om de taal grotendeels onder de knie te krijgen. We moesten dan ook een keuze maken tussen de verschillende programmeertalen.

In oorsprong hadden we vier talen: Fortran, C++, Python en Haskell. De keuze voor Fortran behoeft geen uitleg meer. In C++ en Python wordt actief numeriek ontwerp gedaan. Voor C++ is daar ook over gepubliceerd. Over numeriek ontwerp in Haskell hadden we één paper, die aangaf dat met deze taal dingen mogelijk zijn op het vlak van numerieke software, die in de drie andere talen niet mogelijk zijn. Python hebben we laten vallen, omdat over numeriek ontwerp in Python niet zoveel interessante papers bleken gepubliceerd te zijn als over numeriek ontwerp in Haskell. Ook Fortran hebben we laten vallen, omdat C++ meer mogelijkheden biedt om programmeermethodes uit te breiden en nieuwe problemen aan te pakken.

In onze studie hebben we twee belangrijke aandachtspunten gehanteerd. In de eerste plaats verwachten wij van een programmeertaal die geschikt moet zijn voor numeriek ontwerp, dat ze op zo eenvoudig mogelijke manier omgaat met concepten uit de wiskunde. In de tweede plaats willen wij, dat het resulterende programma ook zo snel mogelijk en correct uitgevoerd zal worden. Dit laatste is een aandachtspunt dat vrij essentieel is voor numeriek ontwerp, omdat men in veel gevallen zeer zware bewerkingen wil uitvoeren die soms verschillende dagen in beslag nemen. Dit laatste is in het verleden al zodanig uitvoerig belicht, dat wij menen dat het tijd wordt dat er meer geïnvesteerd wordt in het eleganter schrijven van numerieke toepassingen. Toch kunnen wij niet omheen het aspect efficiëntie, en dit aspect

zal onvermijdelijk aan bod moeten komen.

De IEEE 754 standaard vierde in 2005 zijn 20ste verjaardag. Toch bleek het zeer moeilijk om grondige informatie te vinden over hoe de faciliteiten, die de standaard biedt, gebruikt kunnen worden in de verschillende talen. We besloten daarom om, voor de twee talen die wij gekozen hebben, dit eens op een rij te zetten. Ook Fortran wordt in dit verband besproken, omdat dit de enige en eerste taal is, waar de faciliteiten van de standaard volledig in de standaard van de taal zelf opgenomen zijn. Deze bespreking vindt u in hoofdstuk 1.

Tijdens de aanvangsfase en ook uit vroegere ervaringen bleek dat, kort nadat Java ten tonele verscheen, er een vrij grote interesse was in Java voor numeriek ontwerp. Die interesse blijkt de laatste jaren zo goed als doodgebloed. We kunnen vaststellen dat bijna niemand nog numeriek ontwerp in Java doet. Ook bleek dat niemand ooit de moeite genomen had om de reden van dit falen te bekijken. Dit hebben wij gedaan in sectie 2.6.

Is het gebruik van objectgerichte methodes wel zo positief voor de efficiëntie van numerieke code? In hoofdstuk 2 bekijken we van nabij welke technieken uit het objectgericht paradigma voor efficiëntieproblemen kunnen zorgen en welke oplossingen ervoor bestaan in C++.

Haskell is een taal die doorgaans niet met numeriek ontwerp geassocieerd wordt. Toch bleek dat een aantal mensen er frequent over gepubliceerd hebben. Haskell blijkt een stijl van programmeren te hebben die, op een andere manier dan men doorgaans gewoon is, toelaat numerieke problemen op te lossen. In hoofdstuk 3 hebben we dan ook de concepten op een rij gezet die Haskell biedt om numeriek ontwerp te vereenvoudigen. We geven in dit hoofdstuk ook twee uitgebreide voorbeelden van hoe deze concepten in de praktijk kunnen omgezet worden.

De kenmerken van twee talen uit een verschillend programmeerparadigma naast elkaar plaatsen, is zoiets als appels met peren vergelijken. Om de vergelijking praktisch te houden, hebben we twee numerieke problemen geïmplementeerd in zowel C++ als Haskell en aan de hand daarvan de vergelijking tussen beide talen voor numeriek ontwerp gestaafd. Deze vergelijking vindt u in hoofdstuk 4.

Dankwoord

In de eerste plaats dank ik mijn promotor en assistenten, die bereid waren om mij als thesisstudent te aanvaarden en steeds met raad en daad bij te staan. Zonder jullie was dit eindwerk nooit geweest wat het geworden is.

Dank aan de lectoren die bereid waren om dit eindwerk te lezen.

Dank aan professor Jerczy Karczmarczuk die bereid was de aangepaste code uit zijn paper op te sturen en daar ook commentaar bij te verschaffen.

Dank aan Edward die steeds bereid was om mijn LaTeX-problemen op te lossen en als klankbord te dienen.

Dank aan mijn vader die bereid was mijn teksten te lezen.

Dank aan alle anderen die mij hebben geholpen.

Dank aan mijn vriendin voor de onvoorwaardelijke steun.

Hoofdstuk 1

IEEE 754, de floating point standaard

De standaard wordt beschreven in IEEE [1985]. Daarnaast is er nog het uitstekende document van Goldberg [1991]. Ook Prof. W. Kahan heeft een aantal zeer lezenswaardige teksten over de standaard en zijn implementatie geschreven (Kahan [2005]). Kahan was één van de belangrijkste ontwerpers van de standaard. In het bijzonder Kahan [1997] bevat een schat aan informatie voor iedereen die de faciliteiten van de standaard wil gebruiken. Ook Higham [2002] wijdt een volledig hoofdstuk aan de floating point voorstelling in software.

Informatie aangaande stabiliteit en *juistheid* in de context van numerieke software is een onderzoeksgebied op zich. Een boek als dat van Higham [2002] is wat numerieke lineaire algebra betreft een must. Heel wat software die je niet met numerieke lineaire algebra associeert maakt gebruik van routines uit de numerieke lineaire algebra. Higham geeft een uitgebreide bibliografie die referenties bevat voor analyses van algoritmes die *niet* direct tot de lineaire algebra behoren.

Dit hoofdstuk bespreekt de faciliteiten die in de taal zelf gerealiseerd kunnen worden om de standaard te gebruiken. De wildgroei aan compileropties en bibliotheekfuncties bespreken gaat voorbij aan het doel van dit hoofdstuk.

Wat volgt zijn geen absolute zekerheden en kan in de nabije toekomst nog veranderen. De toestand van IEEE 754 ondersteuning in de meeste programmeertalen is bedroevend en een standaard interface per taal is in de meeste gevallen niet vastgelegd. Eén en ander heeft te maken met het feit, dat men bij het vastleggen van de standaard niet heeft beslist hoe men alles in talen moest implementeren. De IEEE 754 standaard kan zowel volledig in hardware, volledig in software of een mengeling van beide geïmplementeerd worden. Ga telkens grondig na of de mogelijkheden van de standaard die je wil gebruiken al ondersteund worden door het platform en de compiler waarmee je werkt. Aan overdraagbaarheid van code kan je je niet verwachten op dit vlak.

Bepaalde compilers beïnvloeden door optimalisaties soms subtiel of minder subtiel het resultaat van een bewerking zodat die niet 100% aan de IEEE 754 standaard beantwoordt. Daarnaast kan het zijn dat je bepaalde hardwaremogelijkheden gebruikt die sneller werken, maar die ook niet 100% aan de standaard beantwoorden en soms subtiele problemen kunnen veroorzaken. Een voorbeeld hiervan is de **FMA**-instructie op Intel en andere architecturen. Het vermenigvuldigen van een complex getal met zijn toegevoegde kan door gebruik van **FMA** een niet reëel getal geven (Kahan [1997]).

1.1 Los van talen

Op bepaalde architecturen gebeurt de afronding van floating point getallen anders dan je zou verwachten. Meer bepaald op Intel (zowel i32 als i64) gebeuren alle tussentijdse bewerkingen met maximum precisie voor ze finaal afgerond worden en als resultaat teruggegeven worden. Doorgaans is dit goed, maar het kan resultaten geven die je niet verwacht. Bijvoorbeeld het *bepalen* van epsilon van long doubles in C++ met een stukje code als

```
ee = eps = 1.0;
while(ee + 1.0 != 1.0)
{
    eps = ee; ee = eps /2;
}
```

kan hierdoor grondig in de war gestuurd worden.

Controleren in hoeverre hardware en software de standaard correct implementeren is geen overbodige luxe. Een programma als **paranoia** (Kahan et al. [1998]) voert een grondige controle uit van de IEEE 754 implementatie. Er is ook nog **FPV**, gebaseerd op **FPTST** van Schryer bij AT&T Bell Laboratories, van NAG[1]. Daarnaast kan men nog heel wat informatie en andere software over het testen van floating point implementaties vinden bij Beebe [2004].

Bewerkingen met NaN[2], $\pm\infty$ en niet genormaliseerde floating point getallen zijn niet op alle machines even efficiënt. Op de Intel Pentium 4 nemen bewerkingen met die getallen tot 930 cycles in beslag. Voor meer details zie Dawson [1997] en ook Demmel and Li [1993]. Dit is van belang als je van plan bent om exhaustief van deze IEEE 754 mogelijkheid gebruik te maken bij het schrijven van een rekenintensieve toepassing.

1.2 Precisie

Zowat elke taal laat toe om floating point getallen zowel op *normale* (**3.1415**) als op wetenschappelijke wijze te schrijven (**0.31415e1**). Dit is zo in de behandelde talen. Er zijn een aantal verschillen per taal voor het schrijven van types met verschillende precisie.

Voor de volledigheid bespreken we hier ook de ondersteuning voor signaling NaNs, alhoewel Kahan [1997] opmerkt dat signaling NaNs eigenlijk alleen om politieke redenen in de standaard opgenomen zijn en dat niemand ze schijnt te gebruiken. Momenteel loopt een vraag vanuit het IEEE 754 standaard comité of er eigenlijk wel iemand is die die signaling NaNs gebruikt (Kahan and Zuras [2005]). Signaling NaNs genereren een **INVALID**-exceptie telkens er een bewerking op gebeurt. Elke bewerking die een signaling NaN gebruikt en een resultaat produceert resulteert in een quiet NaN. Signaling NaNs zijn geconcipieerd om gebruikt te worden als waarde voor data die ontbreken. We vermoeden sterk dat signaling NaNs uit de standaard zullen verdwijnen.

1.2.1 Fortran 90/95

De IEEE 754 ondersteuning is zeer recent gestandardiseerd in Fortran met JTC1/SC22 [2004]. Fortran is hiermee de eerste taal die volledige IEEE ondersteuning in zijn standaard opneemt. Momenteel is er slechts één Fortran compiler die al de volledige IEEE functionaliteit ondersteunt en dat is die van NAG. In Metcalf and Reid [1999] wordt de functionaliteit ook al

[1] Numerical Algorithms Group

[2] Not a Number

uiteengezet. We verwachten dat op de duur elke Fortran compiler die functionaliteit zal hebben. Wat we hier en in de volgende secties aangaande Fortran zeggen slaat op wat gestandardiseerd is in de Fortran 2003 standaard.

De ondersteuning is opgesplitst over drie intrinsieke modules

- **IEEE_EXCEPTIONS** Bevat de ondersteuning voor excepties.
- **IEEE_ARITHMETIC** Bevat de ondersteuning voor wat invloed heeft op bewerkingen met floating point getallen.
- **IEEE_FEATURES** Bevat de procedures om op te vragen of bepaalde faciliteiten ondersteund worden door het gebruikte systeem.

Als je **IEEE_ARITHMETIC** gebruikt, wordt automatisch ook **IEEE_EXCEPTIONS** gebruikt. Het gebruik van deze modules kan ervoor zorgen dat om diverse redenen de snelheid op bepaalde processoren vertraagt.

In Fortran is een variabele een floating point getal indien bij de declaratie opgegeven wordt dat het van het type **REAL** is. Voor dubbele precisie kun je **DOUBLE PRECISION** opgeven.

In Fortran kan de precisie bij **REAL** aangegeven worden met de **KIND**-parameter achter **REAL**. **IEEE_SELECTED_REAL_KIND** geeft een waarde terug die ervoor zorgt dat als ze met **KIND** gebruikt wordt, **KIND** minimum de gewenste precisie en exponentgrootte geeft.

```
integer, parameter :: dp = IEEE_SELECTED_REAL_KIND(15,307)
```

Als je met **REAL (KIND=dp) :: i** een variabele declareert, zal **i** minstens 15 decimale cijfers in de mantisse hebben en een exponent hebben die van −307 tot +307 kan gaan. Als het type niet beschikbaar is, dan zal de compilatie afgebroken worden. Bij compilers die de nieuwe standaard nog niet ondersteunen kan je **SELECTED_REAL_KIND** gebruiken in plaats van **IEEE_SELECTED_REAL_KIND**.

Bij veel compilers en platformen kan je de **KIND**-waarde in de handleiding vinden. In veel gevallen staat 4 voor gewone precisie, 8 voor dubbele en 16 voor extended. Je kan en mag daar niet op betrouwen want het staat niet in de standaard.

Voor de speciale IEEE 754 getallen kan je **IEEE_VALUE** gebruiken. **IEEE_VALUE** heeft twee parameters. Het eerste is een variabele of constante van het type waarvan je het resultaat wil hebben. Het tweede argument is één van de volgende:

- **IEEE_SIGNALING_NAN** of **IEEE_QUIET_NAN**: hiervoor moet **IEEE_SUPPORT_NAN** waar zijn
- **IEEE_NEGATIVE_INF** of **IEEE_POSITIVE_INF**: hiervoor moet **IEEE_SUPPORT_INF** waar zijn
- **IEEE_POSITIVE_DENORMAL** of **IEEE_NEGATIVE_DENORMAL**: hiervoor moet **IEEE_SUPPORT_DENORMAL** waar zijn
- **IEEE_NEGATIVE_ZERO** of **IEEE_NEGATIVE_ZERO**

Het rechtstreeks neerschrijven in de taal zonder een aanroep is niet mogelijk.

De **IEEE_SUPPORT**-functies gaan na of voor een bepaald type het speciale soort getal ondersteund wordt.

Bij het schrijven van constanten kan je het **KIND**-type specifiëren door **_kindtype** achter de constante te plaatsen. Als we bijvoorbeeld de constante **1.7_dp** schrijven heeft dit dezelfde

kenmerken als die we daarstraks gedefinieerd hebben met **IEEE_SELECTED_REAL_KIND**. **0.31415E1_8** geeft waarschijnlijk dubbele precisie, maar zeker is dit niet.

1.2.2 C++

C++ kent drie types voor floating point getallen:

- **float**
- **double**
- **long double**

Niet alle compilers hebben **long double**. Je kan testen op de aanwezigheid met **#ifdef HAVE_LONG_DOUBLE**. Pas op: bij sommige compilers hebben **long double** en **double** dezelfde precisie, bijvoorbeeld bij Microsoft's Visual C++ compiler. Bij sommige compilers is er ook zoiets als een **long long double** wat dan een quad precisie floating point getal moet geven maar dit is nergens in de standaard gespecifieerd. Daarnaast blijkt het bij de GNU C++ compiler verkeerdelijk als een **long long** geïnterpreteerd te worden en helemaal geen floating point getal.

De speciale IEEE 754 getallen (quiet/signaling NaN, etc.) kunnen allemaal opgeroepen worden met **numeric_limits**. Zie in sectie 1.3.2 op pagina 13. Ze rechtstreeks neerschrijven in de taal zonder een functie-aanroep is net als in Fortran niet mogelijk.

Bij het schrijven van constanten kan je **l** of **L** achter de constante plaatsen. Dit geeft respectievelijk aan of de constante dubbele of uitgebreide precisie heeft: **0.31415E1L**.

1.2.3 Haskell

Haskell heeft twee floating point types: **Float** en **Double** met respectievelijk enkele en dubbele precisie. De speciale floating point getallen kunnen niet expliciet geschreven worden in Haskell. Je kan ze wel genereren met een bewerking:

- Not-A-Number: **0/0 = nan**
- Oneindig: **1/0 = inf** en **-1/0 = inf**
- Negatieve nul: **-0.0**

Signaling NaNs bestaan niet in Haskell.

1.3 Representatie

1.3.1 Fortran 90/95

Fortran biedt een aantal functies om informatie aangaande floating point getallen te verkrijgen. Je gebruikt telkens een variabele van het type waarvan je informatie wil bekomen als argument. Zie het overzicht in tabel 1.1 op pagina 12.

Fortran biedt ook nog volgende functies om gedetailleerdere informatie te geven:

- **EXPONENT** geeft de exponent van het argument.
- **SET_EXPONENT** zet de exponent.
- **FRACTION** geeft de inhoud van de mantisse.
- **SPACING** geeft de spatiëring van de getallen rond het gegeven getal.

Tabel 1.1: Overzicht van gelijkaardige functies in Fortran, C++ en Haskell

Fortran	C++	Haskell	geeft
DIGITS	**digits**	**floatDigits**	aantal significante getallen
	digits10		decimale precisie
EPSILON	**epsilon()**		machine epsilon
HUGE	**max()**		grootste getal dat voorgesteld kan worden
TINY	**min()**		kleinste getal dat voorgesteld kan worden
MAXEXPONENT	**max_exponent**	**floatRange**	kleinst mogelijke exponent
MINEXPONENT	**min_exponent**	**floatRange**	grootst mogelijke exponent
PRECISION			decimale precisie
RADIX		**floatRadix**	basis van de getallenvoorstelling
RANGE			verschil tussen grootste en kleinste exponent

- **RRSPACING** geeft de inverse van de spatiëring van de getallen rond het gegeven getal.
- **NEAREST** geeft het getal het dichtst in de buurt van het gegeven getal in de richting van een tweede gegeven getal.

Er zijn ook een aantal functies om te testen of een gegeven variabele een speciaal getal is:

- **IEEE_IS_FINITE** om na te gaan of het argument eindig is.
- **IEEE_IS_NAN** om na te gaan of het argument een NaN is.
- **IEEE_IS_NEGATIVE** om na te gaan of een getal negatief is.
- **IEEE_IS_NORMAL** om na te gaan of een getal in normale voorstelling is.

De afrondingsmodus kan opgevraagd worden met **IEEE_GET_ROUNDING_MODE**. Ze kan gezet worden met **IEEE_SET_ROUNDING_MODE** tenminste als dit ondersteund wordt door de processor. Dit kan je nagaan met **IEEE_SUPPORT_ROUNDING**. Met de functie **IEEE_SUPPORT_IO** kan je nagaan of de afronding die gedaan wordt om een floating point getal te tonen, in decimale voorstelling, is volgens wat de IEEE standaard voorschrijft. De vier modi die je kan gebruiken zijn:

- **IEEE_NEAREST** rondt af naar het dichtstbijzijnde getal
- **IEEE_TO_ZERO** rondt naar het dichtsbijzijnde getal dat het dichtst bij nul ligt
- **IEEE_UP** rondt af naar boven
- **IEEE_DOWN** rondt af naar beneden

Op bepaalde machines kan je instellen of *underflow* gebeurt of niet. Dit betekent dat een getal dat te klein is om in genormaliseerde toestand voorgesteld te worden in gedenormaliseerde toestand voorgesteld wordt. Dit kan met **IEEE_SET_UNDERFLOW_MODE** tenminste als **IEEE_SUPPORT_UNDERFLOW_MODE** waar is. Het argument van **IEEE_SET_UNDERFLOW_MODE** is altijd **.FALSE.** of **.TRUE.**. De modus kan je opvragen met **IEEE_GET_UNDERFLOW_MODE**.

De standaard specifieert nog een aantal andere functies. Zie hiervoor in sectie 14.10 van de standaard (JTC1/SC22 [2004]).

1.3.2 C++

C++ biedt de header **<limits>** waarmee je informatie voor elk type kan opvragen (niet alleen voor floating point getallen). Je vraagt de informatie op met **numeric_limits<type>**. Bijvoorbeeld het aantal cijfers in de mantisse van **float**: **numeric_limits<float>::digits**.

De details kan je in het bestand **limits** vinden. Een overzicht vind je in figuur 1.1 op pagina 12.

Er bestaan ook nog een aantal functies voor NaN's:

- **quiet_NaN()** geeft een quiet NaN terug indien die voorgesteld kan worden (te controleren met de functie **has_quiet_NaN**).
- **signaling_NaN()** geeft een signaling NaN terug indien die voorgesteld kan worden (te controleren met **has_signaling_NaN**).

Het blijkt dat je de waarde van **traps** en **round_style**[3] niet kunt betrouwen. De reden hiervoor is, dat controle over trapping en dergelijke niet in de taal ingebouwd is. Voor meer informatie zie sectie 1.4.3 op pagina 16. Enige voorzichtigheid is geboden bij het gebruiken van de informatie die door **numeric_limits** teruggegeven wordt. Je moet voor de correctheid van deze gegevens betrouwen op de zorg die de compilermakers aan hun header files besteed hebben.

1.3.3 Haskell

Haskell biedt, naast wat in figuur 1.1 op pagina 12 staat, volgende functies om informatie over een floating point getal te bekomen:

- **decodeFloat** ontbindt het gegeven floating point getal in mantisse en exponent.
- **encodeFloat** doet het omgekeerde als **decodeFloat**.
- **exponent** geeft de exponent terug.
- **significand** geeft het meest significante getal uit de voorstelling terug.
- **isNaN**, **isInfinite**, **isDenormalized**, **isNegativeZero** en **isIEEE** geeft terug of het een NaN, oneindig, gedenormaliseerd, negatieve nul of IEEE getal is.

1.4 Excepties

De vijf IEEE excepties kunnen op twee manieren gesignaleerd worden:

- Door het zetten van een status vlag
- Door het onderbreken van de uitvoering

Het laatste noemen we, in overeenstemming met de standaard, een *trap*. De exceptievlaggen blijven gezet tot ze expliciet gewist worden. Er is een onderscheid tussen het genereren van een exceptie en een trap. Als een exceptie gegenereerd wordt, wordt de overeenkomstige exceptievlag gezet. Als het trappen voor de exceptie in kwestie aanstaat, wordt de uitvoering van het programma onderbroken en eventueel overgedragen aan een speciaal hiervoor geschreven stuk code. Als dit speciaal stuk code er niet is, stopt het programma. Je moet excepties in deze context onderscheiden van excepties zoals ze bijvoorbeeld in C++ en Java gebruikt worden. Floating point excepties die trappen kunnen bijvoorbeeld niet opgevangen

[3] Die zouden respectievelijk moeten aangeven of traps aanstaan en welke afrondingsmodus gebruikt wordt.

worden met een **catch** block in C++. Een dergelijk systeem heeft men overwogen bij de implementatie van de standaard in Fortran, maar wegens efficiëntieredenen heeft men dit uiteindelijk niet gedaan (Metcalf and Reid [1999]).

De IEEE 754 standaard kent vijf excepties:

- Invalid Operation: resulteert zonder trap in een *quiet NaN*. Bijvoorbeeld bij operaties als 0/0. Bij de Microsoft C en C++ compilers wordt de invalid vlag niet gezet bij het genereren van een NaN (Kahan [1997]).
- Division by Zero: resulteert zonder trap in $\pm\infty$.
- Overflow: resulteert zonder trap in $\pm\infty$.
- Underflow: resulteert zonder trap in ± 0.
- Inexact: wordt gegenereerd als het resultaat van een bewerking niet exact is.

De laatste exceptie is de meest voorkomende en wordt bijvoorbeeld al bij een deling van 1 door 3 gegenereerd.

De trap mode staat normaal af tenzij je die expliciet aanzet. Als je traps aanzet het kan zijn dat routines uit bibliotheken, die je gebruikt, de uitvoering van het programma onderbreken. Sommige routines maken gebruik van het zetten van floating point exceptievlaggen om hun uitvoering aan te passen. Indien je traps aanzet kan de routine en dus het programma onderbroken worden zonder dat er effectief iets fout gegaan is.

Het moet volgens de IEEE 754 standaard eventueel ook mogelijk zijn om een trap opvangroutine te schrijven. Meer dan dat die mogelijkheid er moet zijn zegt de standaard niet. Er is wat dat betreft geen standaardisatie. Je zal de documentatie van de compiler en de machine die je gebruikt moeten raadplegen.

Het (vroeg of laat) opmerken van excepties is cruciaal voor het ontdekken van fouten in het programma en/of het algoritme. Ondersteuning hiervoor wordt helaas zelden in de taal zelf opgenomen maar meestal alleen aan de hand van header files of het zetten van processor vlaggen. Daarnaast geven sommige compilers ook compileeropties om de initiële trap status te veranderen.

1.4.1 Gebruik

Het gebruik van excepties die niet onderbreken is zeer nuttig in volgend geval. Je hebt twee algoritmes:

- één dat altijd correct is maar niet zo snel
- één dat bijna altijd correct is en zeer snel

Als je aan de hand van exceptievlaggen kan nagaan of het snelle algoritme niet correct is ligt volgende methode van oplossing voor de hand:

- Bewaar de vlaggen en zet alle vlaggen op nul.
- Voer het snelle algoritme uit.
- Als geen exceptievlaggen gezet zijn die aangeven dat het algoritme een fout heeft begaan, geef dan het resultaat van het snelle algoritme en herstel de bewaarde vlaggen.
- Als uit de exceptievlaggen blijkt dat er wel een fout is opgetreden, voer dan het trage algoritme uit en herstel de bewaarde vlaggen.

Deze methode heeft maar zin als het snelle algoritme veel sneller is dan het trage, anders kan je net zo goed het trage uitvoeren. Daarnaast is het ook van belang dat op de architectuur waarop je het algoritme uitvoert bewerkingen met NaNs, oneindige getallen en subnormale getallen even snel gebeuren als met andere getallen. Anders kan het snelle algoritme in geval van een fout teveel vertraagd worden omdat een deze methode nog zin zou hebben (Demmel and Li [1993]). Dit zal maar gebeuren als het snelle algoritme geen correct resultaat afgeeft wat meestal niet het geval zal zijn. In dat geval kan je de trapping aanleggen en zelf een opvangroutine schrijven, zodanig dat het snelle algoritme onderbroken wordt in geval van een fout. Omdat dit laatste vervelend is zal dit zelden of nooit gedaan worden.

Het voorbeeld dat Higham [2002] aanhaalt komt uit Demmel and Li [1993]. De LAPACK normschatter om $\|A^{-1}\|_1$ te schatten maakt gebruik van de oplossing van lineaire systemen van de vorm $Ax = b$ en $A^T y = c$. Dit wordt normaal opgelost aan de hand van LU factorisatie $PA = LU$. De oplossing van het triangulaire systeem dat je hieruit verkrijgt kan tot overflow of deling door nul leiden. Daarom gebruikt men in de LAPACK normschatter de routine **xLATRS** die uitgebreide tests en scaleringen doet om overflow en deling door nul te voorkomen. Met gebruik van exceptievlaggen kan je de BLAS-routine **xTRSV** gebruiken en na uitvoering controleren of er excepties gebeurd zijn. Indien overflow gebeurd is moet je alleen nog controleren of dit het gevolg is van het feit dat $\|A\|_1$ te klein is.

Uit de tests die Demmel and Li [1993] uitvoerden blijkt, dat de op deze manier aangepaste normschatters **xxxCON** in LAPACK ofwel 6 keer sneller zijn of 13 keer trager. In het laatste geval was dit het gevolg van NaNs die zich doorheen de stelsels verspreidden op een computer die operaties met NaNs 80 keer trager uitvoert.

In Kahan [1997] vind je per exceptie nog andere goede voorbeelden voor het gebruiken van excepties.

Het constructief gebruiken van traps is veel moeilijker en wordt zelden of nooit gedaan. Je moet speciaal hiervoor een opvangroutine schrijven om ervoor te zorgen dat het programma niet gestopt wordt. De afhandeling van zo'n routine is nogal ingewikkeld. In alle gevallen blijkt het gebruik van traps in de praktijk beperkt te worden tot een hulpmiddel bij het debuggen. Vandaar dat veel compilers het aanzetten van een aantal traps als optie bij het compileren hebben. Hieruit kun je afleiden dat dit ook de voornaamste reden is waarom men niet de moeite heeft genomen het gebruik van traps in de meeste talen vast te leggen in de specificatie van de taal zelf. Uit het slot van Kahan and Zuras [2005] blijkt zelfs, dat men van plan is traps uit de standaard te halen.

1.4.2 Fortran 90/95

De afhandeling van excepties is opgenomen in de module **IEEE_EXCEPTIONS**. In de standaard (JTC1/SC22 [2004]) kan je ook een aantal regels vinden over wanneer en hoe vlaggen door intrensieke Fortran procedures gezet kunnen worden.

Vlaggen

Met de procedure **IEEE_SUPPORT_FLAG** ga je na of een bepaalde IEEE exceptie ondersteund wordt. Het eerste argument is de vlag die je wil controleren. Optioneel kan het tweede argument een variabele van het type **REAL** meegegeven worden waarvan je wil weten of de vlag in kwestie ondersteund wordt voor dat type.

IEEE_GET_STATUS slaat de volledige status van de floating point omgeving op en

IEEE_SET_STATUS herstelt die terug. De namen van de respectieve vlaggen die je kan gebruiken zijn **IEEE_INVALID**, **IEEE_OVERFLOW**, **IEEE_DIVIDE_BY_ZERO**, **IEEE_UNDERFLOW** en **IEEE_INEXACT**.

IEEE_GET_FLAG vraagt de waarde van een bepaalde exceptievlag op. Het eerste argument is de vlag die je wil opvragen (bijvoorbeeld **IEEE_DIVIDE_BY_ZERO**) en in het tweede argument wordt die waarde gezet.

IEEE_SET_FLAG zet de waarde van de IEEE vlaggen. Het eerste argument is de vlag die men wil zetten, het tweede de waarde (**.TRUE.** of **.FALSE.**).

Traps

IEEE_SUPPORT_HALTING gaat na of de processor toelaat om tijdens de uitvoering traps aan of af te zetten voor de gegeven vlag. **IEEE_GET_HALTING_MODE** gaat na of traps aanstaan voor een gegeven vlag. Het eerste argument is de vlag in kwestie, het tweede de variabele waarin het resultaat zal komen.

IEEE_SET_HALTING_MODE zet de waarde van de traps voor een specifieke exceptie op of af. Je kan dit op de volgende manier gebruiken:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
LOGICAL HALTING
...
CALL IEEE_GET_HALTING_MODE(IEEE_OVERFLOW,HALTING)
      ! sla de trapmode op in HALTING
CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,.TRUE.)
      ! zet trapping aan
...
CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,HALTING)
      ! herstel de trapmode voor overflows
```

1.4.3 C99/C++

We herhalen hier voor de duidelijkheid dat floating point excepties *niet* overeenkomen met het exceptiemechanisme dat in C++ bestaat.

De C standaard C99 (ISO [1999]) biedt een aantal functies waarmee afronding en exceptievlaggen gezet en gewist kunnen worden met behulp van de header **fenv.h**. Normaal ondersteunt elke C++ compiler ook dit deel van de C99 standaard. Uitzonderingen zijn de compilers van Borland en Microsoft. Microsoft heeft beloofd om het floating point gedeelte van de C99 standaard op te nemen in Visual C++ 2005. We hebben daar verder niets meer over vernomen. We bespreken vrij kort de mogelijkheden die **fenv.h** biedt. Details zijn te vinden in *The single UNIX Specification*(IEEE [2004]) en in sectie 7.6 van de C99 standaard.

De headerfile **fenv.h** biedt de volgende functies die toelaten om de volledige floating point instellingen op te slaan en te zetten:

- **fegetenv**: vraagt alle instellingen op.
- **fesetenv**: zet de instellingen.
- **feupdateenv**: update de instellingen, alle vlaggen die al gezet waren voor de aanroep blijven gezet na de aanroep van **feupdateenv**.
- **feholdexcept**: doet hetzelfde als **fegetenv** maar wist alle exceptievlaggen en zet alle *traps* af.

Al deze functies hebben slechts één argument: een pointer naar een structuur **fenv_t**. De normale instellingen aan het begin van de programmauitvoeringen zit in de macro **FE_DFL _ENV**. Je kan hier altijd naar terugkeren door **fesetenv(FE_DFL_ENV)** aan te roepen.

Vlaggen

Met **feclearexcept** , **feraiseexcept**, **fesetexceptflag**, **fegetexceptflag** en **fetestexcept** kunnen de exceptievlaggen respectievelijk gewist, gegooid[4], gezet, opgevraagd en getest[5] worden. De macros **FE_DIVBYZERO**, **FE_INEXACT**, **FE_INVALID**, **FE_OVERFLOW**, **FE_UNDERFLOW** en **FE_ALL_EXCEPT** staan respectievelijk voor alle afzonderlijke exceptiebits en voor alle exceptiebits tesamen.

Met de functies **fegetround** en **fesetround** kan je de afrondingsmodus opvragen of zetten.

Traps

De C99 standaard geeft aan dat een systeem eventueel een trapping mode kan leveren in de vorm van een macro die je aan **fesetenv** kan meegeven. Indien dit op een bepaald systeem niet het geval is, zal je niet anders kunnen dan een eventueel geleverde compileroptie gebruiken om traps aan te zetten ofwel rechtstreeks de exceptievlaggen te zetten in de processor. Het type **fenv_t** hoeft niet op alle implementaties hetzelfde te zijn en is ook niet gespecifieerd in de standaard.

Indien je alleen met een compileroptie traps kan aanzetten, kan je gebruik maken van **feholdexcept** om alle traps af te zetten. Je kan daarna eventueel de traps opzetten met **fesetenv** als je de status voor het aanroepen van **feholdexcept** hebt opgeslagen met **feget env**.

Heb je de beschikking over de GNU C Library (Glibc: GNU [2005]) dan kunnen traps op alle excepties gezet worden met **fesetenv(FE_NOMASK_ENV)**. Je kan controleren met **#ifdef FE_NOMASK_ENV** of dit wel degelijk mogelijk is.

Glibc definieert ook de functies **feenableexcept**, **fedisableexcept** en **fegetexcept** die toelaten om afzonderlijke traps op excepties aan en af te zetten en de status van het gooien van traps op te vragen.

1.4.4 Haskell

Haskell heeft niet de mogelijkheid om excepties te trappen noch om exceptievlaggen op te vragen.

1.5 Conclusie

De meest consequente ondersteuning van de IEEE standaard kan je vinden in de Fortran 2003 standaard. Dit is ook logisch, omdat het de enige taal is die de standaard volledig ondersteunt en omdat Fortran bijna uitsluitend gebruikt wordt voor numeriek ontwerp. In de andere talen die wij bekeken hebben is de ondersteuning noch consequent noch volledig.

In C++ moet je gebruik maken van twee verschillende standaarden, die van C en die van C++. Dit is voor een stuk normaal omdat C++ ook grotendeels de C standaard volgt. Men

[4]indien traps niet opstaan heeft dit alleen als gevolg dat de exceptievlag aangezet wordt

[5]Als traps opstaan voor de exceptie waarvoor je test zal er getrapt worden.

gaat wat in de C standaard opgenomen is niet nog eens overdoen voor de C++ standaard als daar geen goede redenen voor zijn. Desondanks is het wel jammer dat trapping excepties niet kunnen omgezet naar *echte* C++ excepties. Merk op dat een vergelijkbare ondersteuning van floating point excepties voor Fortran overwogen is, maar omwille van efficiëntieredenen en problemen met de consequentie van code verworpen is (Metcalf and Reid [1999]).

Dat de volledige ondersteuning in de meeste talen helemaal geen feit is of bijna 20 jaar op zich heeft laten wachten, heeft de ontwikkeling van numerieke software helemaal niet tegengehouden. Je kunt je de vraag stellen of het überhaupt zin heeft om al de faciliteiten die de standaard vraagt, te bieden in een taal. Wij vinden van wel. Het gebrek aan consequente ondersteuning heeft het de ontwikkelaars van bibliotheken niet eenvoudig gemaakt. Als de standaard van in het begin consequent door talen was ondersteund, dan was zeker heel wat moeite bespaard in de ontwikkeling van numerieke software de laatste tientallen jaren. Daarnaast heeft het ontbreken van die faciliteiten er voor gezorgd, dat de meesten ze niet of slechts gedeeltelijk kennen en ook geen flauw idee hebben van de mogelijkheden die ze bieden. Meer nog, door de wirwar aan opties word je meestal afgeschrikt om er gebruik van te maken, omdat je vreest in de problemen te komen. Doordat niet iedereen de standaard ten volle benutte was er ook geen stimulans voor hardware-ontwikkelaars om de standaard correct en goed te implementeren.

Samenvattend zijn de twee redenen die wij zien voor het uitblijven van een volledige ondersteuning van de standaard in een taal de volgende. Ten eerste specifieert de standaard van bepaalde faciliteiten niet of ze nu wel in hardware, in software of in een combinatie van beide geïmplementeerd moeten worden. Ten tweede is naar bepaalde faciliteiten weinig vraag. Zo heeft het zelfs voor een taal als Fortran, die uitsluitend gericht is op numeriek ontwerp, bijna twintig jaar geduurd voor volledige ondersteuning in de standaard opgenomen is. Momenteel overweegt men zelfs om een aantal van de weinig gebruikte faciliteiten uit de standaard te halen (Kahan and Zuras [2005]).

Hoofdstuk 2

Objectgericht ontwerp en numerieke toepassingen

"Beauty and utility are not synonymous." *Slashdot lezer*

2.1 Inleiding

Het gebruik van objectgerichte talen voor numerieke toepassingen mag voor veel informatici logisch lijken. Dit is *niet* het geval. Het merendeel van de algoritmen klaar voor implementatie is procedureel gestructureerd. Je gaat als niet programmeur of beginnend programmeur niet nog eens de last op je nemen om een objectgerichte taal onder de knie te krijgen als daar op het eerste zicht geen goede redenen voor zijn.

Als je je het objectgericht denken *wel* eigen hebt gemaakt, kan je de vraag stellen of het überhaupt zin heeft om een objectgerichte taal voor numerieke toepassingen te gebruiken. We weten wel dat objectgericht ontwerp de ontwikkeling en het onderhoud van software vereenvoudigt. Is het voor numerieke toepassingen ook efficiënt?

Je kan twee stijlen hanteren:

- Je gebruikt een gemengde stijl. Voor het algemene ontwerp van het programma gebruik je alle functionaliteit die objectgericht ontwerp biedt. Het zware rekenwerk laat je over aan procedureel gestructureerde code en bibliotheekaanroepen naar bijvoorbeeld BLAS-kernels. Zodoende vermijd je de abstractiekost van een objectgerichte taal en krijg je het karakter van een scriptingtaal zoals Python.
- In bepaalde toepassingen is het gebruik van procedurele code en externe bibliotheekaanroepen niet mogelijk of zelfs helemaal niet wenselijk. Dan zal je *wel* gebruik maken van de mogelijkheden die een objectgerichte taal biedt in kritieke code[1].

De eerste stijl behandelen we niet. Wat de tweede stijl betreft zullen we de verschillende mogelijke efficiëntieproblemen bekijken, die kunnen optreden bij het gebruik van een objectgerichte taal als C++, samen met hun mogelijke oplossingen. Gaandeweg zullen we een aantal beperkingen opleggen om efficiëntie te behouden in kritieke code. Tegelijkertijd zullen we ook kijken wat de impact is op de eenvoud van het schrijven van numerieke code.

[1]Wij bedoelen hiermee code die zo efficiënt mogelijk moet zijn omdat ze zeer intensief gebruikt wordt. Een kleine verbetering in snelheid van die code heeft een grote impact op de snelheid van het volledige programma.

2.2 Architectuur

2.2.1 Inlining

Door het opsplitsen van een probleem in zijn functionele componenten krijg je een vertraging van de code ten opzichte van niet opgesplitste code. De vertraging wordt veroorzaakt doordat men telkens een functie of methode moet oproepen. Inlining is het vervangen van de aanroep van een methode door de code in de methode zelf. Het gebruik van inlining in C++ is cruciaal voor het verkrijgen van snelle code.

Inlinen kan op twee manieren:

- Schrijf de code in de definitie van de klasse
- Plaats het sleutelwoord **inline** voor de definitie van de methode

De compiler is altijd vrij om dit *advies* te negeren.
Inlinen heeft volgende voordelen

- Het vermijdt de overhead van het aanroepen van een functie.
- Het geeft de compiler de mogelijkheid om code te optimaliseren die zonder inline gebracht te zijn in een andere methode te ver van mekaar verspreid zou zijn om voor optimalisatie in aanmerking te komen.

Inlinen heeft volgende nadelen

- De grootte van het uitvoerbare bestand kan aanzienlijk worden groeien.
- De omvang van de actieve code kan te groot worden voor de cache en bijgevolg voor cache misses zorgen en een negatieve impact hebben op de performantie. Vermijd dit.

Het eerste nadeel is niet zo'n probleem. Het tweede is veel ingrijpender. Let goed op dat niet al te veel niet-kritieke code geïnlined wordt in kritieke code.

Inlinen gebruik je sowieso in volgende gevallen

- Voor triviale methodes en *singletons*
- Voor veel gebruikte methodes

In het eerste geval zal deze optimalisatie meestal door de compiler gebeuren als optimalisatie ingeschakeld is. Methodes waarvan je denkt dat het zin heeft ze te inlinen kunnen soms een negatieve impact hebben op de performantie. Controleer daarom altijd of het inlinen van een stuk code wel een positieve impact op de snelheid heeft. Dit doe je best met een profiler.

Om het teveel aan cache misses te beperken, kan het zin hebben een grote methode met veel code, waarvan meestal maar een klein stukje wordt uitgevoerd, te inlinen, maar het grote stuk dat bijna nooit wordt uitgevoerd te verplaatsen naar een andere methode. Die methode wordt dan opgeroepen in de oorspronkelijke methode. Je vermijdt zo dat code die zelden of nooit uitgevoerd wordt een impact heeft op de grootte van de actieve code en onnodig voor cache misses zorgt.

Callback inlining

Een callback is een functie die, meegegeven aan een andere functie, door die andere functie aangeroepen kan worden. Callbacks komen veelvuldig voor in numerieke code. Een integratiemethode gebruikt een callback naar de functie die de gebruiker meegeeft en waar de

Figuur 2.1: Callback inlining

(a)

```
double functie(double x)
{
    ...
}

template<double f(double)> double integreer(double start, double einde)
{
    ...
        temp += f(p);
    ...
}
```

(b)

```
resultaat = integreer<functie> (1.0,2.0);
```

methode de integratie op uitvoert. Ze kunnen klassiek op een aantal verschillende manieren geïmplementeerd worden

- Een pointer naar een functie. Een typische oplossing in C.
- Als virtuele functie in een klasse die vervolgens afgeleid wordt om een concrete versie van de functie die zal moeten aangeroepen worden te implementeren. Dit is een typische oplossing in C++.

Het vervelende aan deze oplossingen is dat de functie niet inline kan gebracht worden. Als de functie klein is en veel opgeroepen wordt, zal dit een negatieve impact op de performantie hebben.

Er bestaan twee oplossingen met behulp van templates die ervoor zorgen dat je die functie wel kan inlinen

- Je plaatst de functie in een klasse die de ()-operator definieert en geeft die klasse als template parameter mee aan de concrete klasse die die functie als callback zal moeten aanroepen.
- Je geeft de functie als pointer naar een functie als template parameter mee.

De laatste oplossing heeft als voordeel dat je niet telkens een nieuwe klasse moet definiëren als je een functie als argument wil doorgeven. Deze oplossing wordt geschreven en gebruikt als de code in figuur 2.1a en 2.1b op pagina 21.

2.2.2 Aliasing

Aliasing is een vervelend probleem in C++. Grote structuren (matrices bijvoorbeeld) worden doorgegeven als pointers en de bewerkingen op die structuren kunnen onderhevig zijn aan aliasing. Dit maakt een aantal belangrijke compileroptimalisaties onmogelijk. Dit kan een negatieve weerslag van 20% tot 50% op de performantie hebben van loops die volledig in de cache passen (Veldhuizen [1999]). Goede compilers zullen aliasanalyse doen, maar dit lost

niet alle problemen op. Veel compilers ondersteunen het sleutelwoord **restrict** om aan te geven dat voor een gegeven pointer naar een set data geen aliases bestaan. Bijvoorbeeld:

```
void add(double* restrict a, double* restrict b, int n)
{
    for (int i = 0; i < n; i++) a[i] += b[i];
}
```

De structuur waar **a** en **b** naar verwijzen bevatten geen aliases en de compiler mag er dan bij de optimalisatie vanuit gaan dat **b** geen stukken van **a** bevat en omgekeerd. Het sleutelwoord **restrict** is geïntroduceerd door NCEG (Numerical C Extensions Group) en is ondertussen opgenomen in de nieuwe ANSI/ISO C standaard. Het zal waarschijnlijk in de volgende C++ standaard opgenomen worden.

2.3 Klassen

2.3.1 Methodes

Het rechtstreeks aanroepen van een niet virtuele methode op een object kost niet meer dan het aanroepen van een normale functie (Goldthwaite [2003]). Het verpakken van een functie in een klasse geeft bijgevolg geen overhead.

2.3.2 Encapsulatie

Een belangrijk voordeel aan het gebruik van encapsulatie is, dat objecten variabelen groeperen in één structuur. Dit kan voordelig zijn voor de efficiëntie van de uitvoering omdat dit de lokaliteit verbetert in het geheugen en dit is beter voor cache misses (Robison [1996]). Dit hangt af van hoe de logica van het programma in mekaar zit. Een programma dat op heel veel objecten van hetzelfde type werkt, zal voordeel hebben als het vooral operaties uitvoert op alle gegevens uit één object. Als het programma vooral operaties uitvoert op één gegeven uit alle objecten, zal dit groeperen een negatieve impact hebben. Tussen elk gegeven dat je nodig hebt zit een groep gegevens die je niet nodig hebt. Door cache misses vertraagt het programma dan.

Je mag encapsulatie in kritieke code niet tot het uiterste drijven en al te veel abstractie gebruiken. Hoe meer het opvragen van data gedelegeerd wordt naar nog eens een geëncapsuleerd object, hoe langer het duurt om de gegevens te bereiken en hoe meer cache misses kunnen optreden. Als de kritieke code inline kan gebracht worden, kan dit probleem soms opgelost worden. Dit is één van de basisideeën die bij expression templates gebruikt worden (zie sectie 2.4.2 op pagina 28). Daar maak je excessief gebruik van het delegeren van verantwoordelijkheden van methodes, maar door het gebruik van inlining geeft dit uiteindelijk geen negatieve weerslag op de performantie.

2.3.3 Overerving

Constructoren en destructoren van een op zichzelf staande klasse creëren doorgaans geen performantieproblemen. Alleen als een constructor van een klasse, die onderaan een grote hiërarchie staat, opgeroepen wordt, resulteert dit in het oproepen van de constructor van alle ouderklassen. Hoe groter de hiërarchie hoe meer onnodige bewerkingen gebeuren en hoe meer overtollige objecten doorgaans gecreëerd worden.

Je doet er best aan om de hiërarchie van objecten die in kritieke code gebruikt worden zo laag mogelijk te houden en grondig te controleren op code die al te algemeen is doorheen de hiërarchie van de gebruikte objecten. Het probleem bij constructoren en destructoren is groter dan bij andere methodes, omdat het aanroepen van constructoren en destructoren bijna altijd gepaard gaat met het creëren van objecten die een dochterklasse niet nodig heeft.

Als je gebruik maakt van reeds bestaande klassen en daar een eigen klasse van afleidt neem je best een kijkje in de code van de klassen waarvan je afleidt. Als de eigen afgeleide klasse in kritieke code gebruikt wordt, is dit zeker geen overbodige luxe.

2.3.4 Excepties

Veelvuldig gooien en opvangen van excepties in kritieke code is geen goed idee. Door de stack unwinding die moet gebeuren bij het gooien van elke exceptie zal alles veel trager verlopen. Vermijd het gebruik van excepties in kritieke code.

2.3.5 Tijdelijke objecten

Als je een objectgerichte taal ondoordacht gebruikt genereer je objecten die helemaal niet gebruikt worden of die alleen maar verder gekopieerd worden in een ander object. Dit heeft als overhead dat per object dat nutteloos gegenereerd wordt een constructor en destructor uitgevoerd wordt. Daarnaast wordt ook een stuk geheugen in gebruik genomen en terug uit gebruik gehaald. Dit zorgt voor heel wat *onzichtbare* objecten die allemaal kostbare tijd in beslag nemen en helemaal geen nuttig werk doen. In kritieke code zijn ze uit den boze. Veel van deze tijdelijke objecten worden weggeoptimaliseerd door de huidige compilers, maar dit is niet altijd en niet bij elke compiler het geval.

Pass by value

Het doorgeven van objecten als waarde en niet met een referentie geeft in de meeste gevallen nodeloos extra werk. Je hebt twee opties in C++:

- Je geeft het object door met een pointer (bv. **Matrix *b**)
- Je geeft het object door en vangt het op met een referentie **Matrix &b**

Het laatste geeft meer leesbare code en maakt dat je geen nul-pointers kan doorgeven die voor fouten in het programma kunnen zorgen. Dit is de geprefereerde methode.

Default constructie

Het lijkt voor zich te spreken om een variabele die een object moet bevatten als volgt te declareren aan het begin van een methode: **Matrix a;**. Dit is een slecht idee. Dit creëert automatisch een object in a. Als je wat verder in de code een andere matrix aan die variabele toekent gaat dit object verloren. Beter is om die variabele pas te declareren op het moment dat het object eraan toegekend wordt: **Matrix a = b;** Dit vermijdt een ongebruikt tijdelijk object. Daarnaast kan het ook gebeuren dat het object **a** helemaal niet gebruikt wordt tijdens de uitvoering van de code. Dan is de creatie van het tijdelijke object in **a** helemaal nutteloos. Als algemene regel geldt: stel de declaratie van een variabele uit tot in de scope waar de variabele absoluut nodig is.

Figuur 2.2: Tijdelijke objecten bij toekenningen vermijden

(a)

```
a = b + c + d + e;
```

(b)

```
a = b;
a+= c;
a+= d;
a+= e;
```

Tijdelijke objecten bij toekenningen

Een toekenning als **a = b + c;** creëert een tijdelijk object waarin **b + c** opgeslagen wordt en vervolgens wordt dit tijdelijk object aan **a** toegekend. Je kan beter **a += c;** gebruiken. Dit vermijdt een tijdelijk object. Er worden meer tijdelijke objecten gecreëerd naarmate je meer bewerkingen aan de rechterkant van een toekenning schrijft. De code in figuur 2.2a genereert bijvoorbeeld 3 tijdelijke objecten. Je kan beter de code in figuur 2.2b schrijven.

Als de +=- en =-operator efficiënt geschreven zijn (met behulp van referenties in plaats van pass by value) zal dit geen tijdelijke objecten genereren en de gebruikelijke overhead met het creëren van tijdelijke objecten vermijden.

Dit gaat *niet* op voor code die gebruik maakt van expression templates (zie sectie 2.4.2 op pagina 28). Dergelijke toekenningen zullen dan, tenminste als de expression templates goed geschreven zijn, juist efficiënter gebeuren op de eerste manier.

Data members en constructie

Men doet er best aan objecten als data members in een klasse te initialiseren *voor* de constructor zijn uitvoering begint. Alle data members die nog niet geïnitialiseerd zijn voor de constructor zijn uitvoering begint, krijgen een default object toegekend (Lippman [1996] en Goldthwaite [2003]). Er zijn twee mogelijkheden

- Men initialiseert het object op de plaats waar het gedeclareerd wordt in de klasse.
- Men initialiseert het object in de initialisatielijst van de constructor.

De eerste mogelijkheid is verboden volgens de ANSI C++ standaard. Men kan beter voor de tweede mogelijkheid kiezen.

Schrijf niet de code in figuur 2.3a op pagina 25 maar die in figuur 2.3b. Dit zal niet altijd mogelijk zijn.

Type conversie

Een functie aanroepen met een argument dat niet van hetzelfde type is kan ervoor zorgen dat extra werk moet worden gedaan tijdens de uitvoering om het type van de variabele aan te passen. Het is beter om alle constructoren met slechts één argument[2] **explicit** te declareren.

[2] Dit geldt ook voor constructoren met meerdere argumenten waarvan er default waarden hebben, zodanig dat die constructoren met één argument opgeroepen kunnen worden.

Figuur 2.3: Initialisatie van data members.

(a)

```
class X
{
    Z var1;
    Z var2;

    X(Z b)
    {
        var1 = 0;
        var2 = b;
    }
}
```

(b)

```
class X
{
    Z var1;
    Z var2;

    X(Z b): var1(0), var2(b)
    {}
}
```

Dit voorkomt dat je accidenteel aan impliciete type conversie[3] doet en in kritieke code tijd verliest met type conversie.

Een typisch voorbeeld is een klasse van complexe getallen zoals in figuur 2.4a. Als de variabele **t** van het type **Complex** is, dan zal **t = 0;** ervoor zorgen dat er eerst een tijdelijk object gecreëerd wordt, waarbij een **int** naar een **double** geconverteerd moet worden. Dit tijdelijk object wordt dan vervolgens toegekend aan **t**. Dit zijn twee overbodige operaties. Daarnaast kan het zijn, dat er ook een constructor bestaat met als argument een **int** die iets helemaal anders betekent dan wat wij veronderstellen als we het schrijven.

Return Value Optimization

Bij code als in figuur 2.4b op pagina 26 kan een compiler meestal de generatie van het tijdelijke object **result** wegoptimaliseren en het resultaat rechtstreeks in het toegekende object plaatsen. Dit heet *return value optimization*. Niet alle compilers doen dit automatisch. Zeker niet als de code complexer is dan dit eenvoudige stukje code. De code in figuur 2.4c signaleert dat door het resultaat expliciet als een nieuwe objectcreatie te schrijven. Bijna alle compilers optimaliseren dit soort code. In het geval de compiler dit niet doet, kan je eventueel *computational constructoren* gebruiken (Bulka and Mayhew [1999]).

[3] Voor meer details aangaande impliciete type conversie zie p. 284 in Stroustrup [2000]

Figuur 2.4: Optimalisatie van return values

(a)

```
class Complex
{
    Complex(double x, double y = 0.0) _x(x),_y(y)
}
```

(b)

```
Complex operator+ (const Complex& a, const Complex& b)
{
    Complex result;
    result.real = a.real + b.real;
    result.imag = a.imag + b.imag;
    return result;
}
```

(c)

```
Complex operator+ (const Complex& a, const Complex& b)
{
    return Complex(a.real+b.real,a.imag+b.imag);
}
```

2.4 Polymorfisme

Polymorfisme in C++ wordt slechts toegepast bij methodes die **virtual** gedeclareerd worden. Het gebruik van virtuele functies heeft als grootste performantiekost dat de code uit de functie niet meer inline[4] kan gebracht worden (Bulka and Mayhew [1999] en Lippman [1996]).

2.4.1 Templates

In plaats van de keuze tussen twee methodes tijdens de uitvoering van het programma te doen kan je in veel gevallen die keuze op het moment van de compilatie maken. Dit heet statisch polymorfisme in tegenstelling tot dynamisch polymorfisme. In C++ gebruik je hiervoor templates om niet teveel aan ontwerpflexibiliteit te verliezen.

De keuze van de methode die uitgevoerd wordt moet je kunnen parametriseren. Je hebt bijvoorbeeld een matrixklasse die een aantal verschillende types kan hebben (bijvoorbeeld symmetrische matrix, bandmatrix, . . .). Je kunt dan een algemene interface opstellen in een basis matrixklasse en voor de specifieke types matrix daarvan afleiden. Je zult telkens in een virtuele functie de specifieke versie van operaties voor dat type definiëren. Gezien dat meestal eenvoudige operaties zijn die veel opgeroepen worden en niet inline gebracht kunnen worden zal dat de code vertragen.

Met gebruik van templates kun je de opslaginformatie per specifieke matrixsoort in een klasse plaatsen en die als template parameter doorgeven aan de matrix klasse. De klasse met

[4] Zie sectie 2.2.1 op 20

```
// De basisklasse neemt een template parameter. Deze parameter
// is het type van de klasse die ervan afgeleid zal worden.
template<class T_leaftype>
class Matrix
{
    public:
        T_leaftype& asLeaf()
        { return static_cast<T_leaftype&>(*this); }

        double operator()(int i, int j)
        { return asLeaf()(i,j); }      // delegeer aan leaf
};

class SymmetricMatrix : public Matrix<SymmetricMatrix>
{
    ...
};

class UpperTriMatrix : public Matrix<UpperTriMatrix>
{
    ...
};

// Voorbeeld routine die elke type Matrix als argument kan nemen
template<class T_leaftype>
double sum(Matrix<T_leaftype>& A);

// Voorbeeld
SymmetricMatrix A;
sum(A);
```

Figuur 2.5: De *Barton and Nackman Trick*

de informatie voor de opslag van de matrix wordt *engine*[5] genoemd. Dit vermijdt het gebruik van virtuele functies.

Deze methode heeft een heel vervelend nadeel. De basisklasse krijgt de unie van alle methodes die alle specifieke types matrices nodig hebben. Als je voor een specifiek type matrix een speciale methode wil toevoegen, dan wordt die methode automatisch ook toegevoegd aan alle andere matrixtypes. Bijvoorbeeld een methode om te controleren of een matrix symmetrisch positief definiet is.

Er bestaat een eenvoudige oplossing hiervoor die men de *Barton and Nackman Trick* noemt (Barton and Nackman [1994]). De truuk bestaat erin, om de basisklasse een template parameter te geven die door de afgeleide klasse zal gebruikt worden om zichzelf door te geven als template parameter. Die template parameter wordt in de basisklasse gebruikt om met **static_cast** alle operaties die uit de afgeleide klasse moeten gehaald worden door te geven aan de methodes van de afgeleide klasse in kwestie. Dit gebeurt bij het compileren. Je kan operaties toevoegen aan de afgeleide klassen die niet bij de andere afgeleide klassen moeten horen. Zie figuur 2.5 op pagina 27.

[5]Terminologie door het POOMA team gebruikt (Veldhuizen [1999]).

2.4.2 Expression templates

Een geavanceerdere vorm van statisch polymorfisme zijn expression templates (Veldhuizen [1995]). Deze techniek laat toe om bepaalde uitdrukkingen zeer dicht bij hun wiskundige formulering te schrijven in C++. Terzelfdertijd is ze ook efficiënt.

We kunnen bijvoorbeeld bij een integratie, de opgegeven functie inlinen in de code voor de integratie

```
// Integrate a function from 0 to 10
DoublePlaceholder x;
double result = integrate(x/(1.0+x), 0.0, 10.0);
```

De uitdrukking voor de functie die moet geïntegreerd worden **x/(1.0+x)** wordt bij compilatie inline in de code van de integratiemethode geplaatst.

Een ander voorbeeld is het vermenigvuldigen van matrices wat eenvoudigweg als volgt geschreven kan worden: **A*B**[6].

Een ander zeer belangrijk voordeel van expression templates is, dat het tijdelijke objecten vermijdt bij bewerkingen van de vorm

```
A = B + C + D;
```

Door alleen operator overloading te gebruiken is dit slechts efficiënt als we de vorige uitdrukking omvormen tot

```
A = B;
A += C;
A += D;
```

Met gebruik van expression templates kan de volledige oorspronkelijke uitdrukking tijdens compilatie omgevormd worden tot

```
for (int i = 0; i < A.size(); i++) A[i] = B[i] + C[i] + D[i];
```

wat veel efficiënter is en waar de compiler loopoptimalisaties op kan uitvoeren.

Expression templates maken gebruik van volgende technieken

- templates
- operator overloading
- overerving
- inlining

Hoe werken expression templates?

Wij tonen hier hoe een Fortran **WHERE** in C++ met expression templates kan uitgewerkt worden. We zullen dit doen voor het stukje Fortran code

```
WHERE (M < 0.0)
    M = 0.0
ENDWHERE
```

Dit stukje code zet alle elementen in de matrix of vector **M** die kleiner zijn dan 0.0 op 0.0. In figuur 2.8 op pagina 32 vindt u het klasse- en interactiediagram die bij deze code horen. In C++ kunnen we met expression templates de code in figuur 2.6a op pagina 29 schrijven.

[6] Pas op, in de meeste implementaties is dit het elementsgewijs vermenigvuldigen van twee matrices en geen matrixvermenigvuldiging.

Figuur 2.6: Fortran **WHERE** expression template code

(a) Code voor het op 0.0 zetten van alle negatieve waarden in **x**.

```
DCompExpr<DCompExprIdentity> x;
vector.where(x < 0.0, 0.0);
```

(b) Code die de waarde **b** zet voor alle waarden van de vector die aan **expr** beantwoorden.

```
template <class CompExpr>
void where(DCompExpr<CompExpr> expr, double b)
{
    for(int i = 0; i < dim; i++)
    {
        if(expr(data[i])) data[i] = b;
    }

}
```

(c) Code van de klasse van de algemene vergelijkingsexpressie.

```
template<class A>
class DCompExpr
{
    A a_;

    public:

    DCompExpr(): a_(A()) {}

    DCompExpr(const A& x):a_(x) {}

    double val(double x) const
    {
        return a_.val(x);
    }

    bool operator()(double x) const
    {
        return a_(x);
    }
};
```

(d) Code voor de operator die een vergelijking omzet in een **DCompExpr**-type.

```
template<class A>
    DCompExpr<DBinCompExprOp<DCompExpr<A>, DCompExprLiteral, DOpLT> >
    operator <(const DCompExpr<A>& a, double b)
{
    typedef DBinCompExprOp<DCompExpr<A>, DCompExprLiteral, DOpLT> CompExprT;
    return DCompExpr<CompExprT>(CompExprT(a, DCompExprLiteral(b)));
}
```

De code voor de methode **where** in de Vector klasse vind je in figuur 2.6b. Elk functioneel onderdeel van de expression template code wordt als template argument **A** verpakt in de klasse **DCompExpr** (zie figuur 2.6c).

Om de uitdrukking **x < 0.0** om te zetten in het type **DCompExpr** moet een **<**-operator gedefinieerd worden (zie figuur 2.6d). **x** wat van het type **DCompExprIdentity** is komt binnen in **a**. De waarde **0.0** (**b**) wordt verpakt in een object van het type **DCompExprLiteral**. De **typedef** definieert het type **CompExprT** wat een algemene binaire vergelijkingsoperator (**DBinCompExprOp** in figuur 2.7a op pagina 2.7a) is. De algemene binaire vergelijksoperator krijgt drie template argumenten: twee subexpressies en een concrete vergelijkingsoperator. De binaire vergelijkingsoperator zal een waarde krijgen en die doorgeven aan zijn twee subexpressies (in dit geval **x** en **0.0**). Het resultaat dat die twee subexpressies teruggeven, wordt dan met de gegeven concrete vergelijkingsoperator (in dit geval de kleiner dan operator **DOpLT** zoals in figuur 2.7b) vergeleken.

In figuur 2.7a zal **a_** de identiteitsuitdrukking zijn en altijd de waarde teruggeven die eraan gegeven wordt. **b_** zal in dit geval de waarde teruggeven die erin is opgeslagen en niet die welke eraan gegeven wordt. Stel nu dat in ons geval bij de **where** code (figuur 2.6b) aan **expr** de waarde -1.0 wordt meegegeven. Dan geeft **a_** hier -1.0 terug en **b_** de waarde die erin is opgeslagen, in dit geval 0.0. De concrete vergelijkingsoperator kan die twee waarden vergelijken en het resultaat teruggeven.

Het feit dat de methode **apply** in **DOpLT** (figuur 2.7b) inline gedeclareerd is, zorgt ervoor dat de vergelijking **a < b** na compilatie inline in de **where**-code terecht zal komen.

De code van de identiteit en de letterlijke waarde is vanzelfsprekend zoals in figuur 2.7c.

Samenvattend ondergaat de uitdrukking **expr(data[i])** in **where** volgende veranderingen tijdens het compileren

- **expr(data[i])**
- **DCompExpr(data[i])**
- **DBinCompExprOp(data[i],0.0,DLTOp)(date[i])**
- **DLTOp.apply(Identity.val(data[i]),Literal(0.0).val(data[i]))**
- **(data[i] < 0.0)**

Wat gebeurt kunnen we ook uitdrukken in een interactiediagram (zie figuur 2.8b). Merk op dat wat in het interactiediagram staat, niet zal gebeuren tijdens de uitvoering van het programma. Door het gebruik van templates en inlining zal dit gebeuren tijdens de compilatie van het programma.

Uit een kleine test (zie figuur 2.9 op pagina 33) blijkt dat onze versie even snel werkt als de standaard C++ versie met iterators. Maar de kans op het schrijven van fouten, eens de template code grondig getest is, is veel kleiner dan bij

```
d_type *iterator = &vector[0];
while (itt != &vector[GROOTTE])
{
    if (*iterator < 0.0) *iterator = 0.0;
    ++iterator;
}
```

De uitdrukking **vector.where(x < 0.0, 0.0)** ligt ook veel dichter bij wat je wilt uitdrukken. Een volledig werkende **WHERE** zoals in Fortran bestaat is mogelijk in C++ maar zal meer programmeerwerk vragen.

Figuur 2.7: Fortran **WHERE** expression template code

(a) Code van de klassie die een operator (**Op**) toepast op twee gegeven waarden (**A** en **B**).

```
template<class A, class B, class Op>
class DBinCompExprOp
{
    const A& a_;
    const B& b_;

    public:
    DBinCompExprOp(const A& a, const B& b)
        : a_(a), b_(b)
        { }

    bool operator()(double x) const
    {
        return Op::apply(a_.val(x), b_.val(x));
    }
};
```

(b) Code van de klasse die een kleiner dan vergelijking uitvoert.

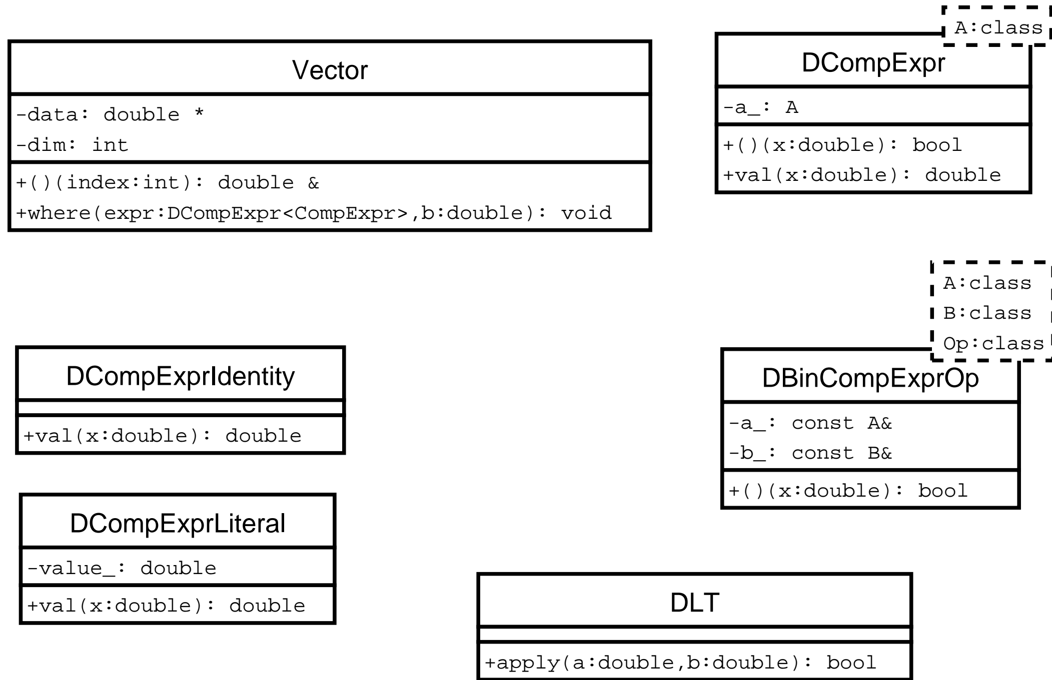
```
class DOpLT
{
    public:
        static inline bool apply(double a, double b)
        {
            return a < b;
        }
};
```

(c) Code voor de identiteit en de letterlijke waarde.

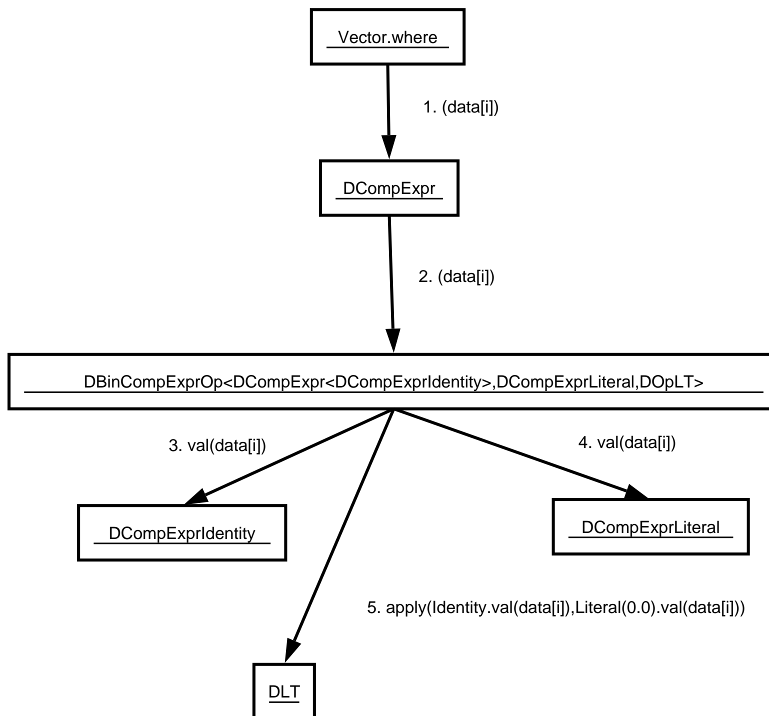
```
class DCompExprIdentity
{
    public:
        DCompExprIdentity()
        { }
        double val(double x) const { return x; }
};

class DCompExprLiteral
{
    double value_;

    public:
    DCompExprLiteral(double value): value_ = value { }

    double val(double x) const { return value_; }
};
```

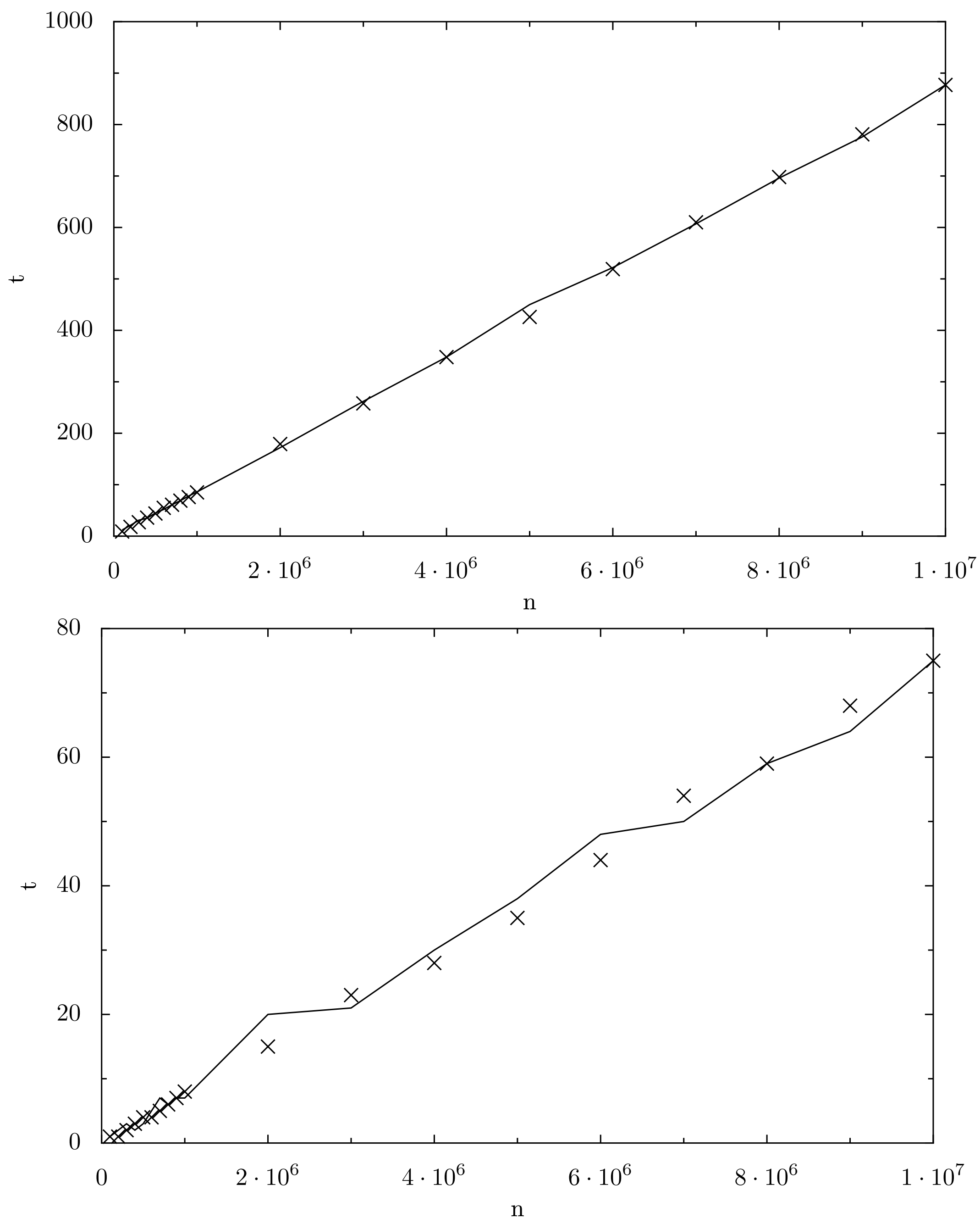
Figuur 2.8: Expression templates

(a) Klassendiagram van de **where**-code in C++.



(b) Interactiediagram van de **where**-code in C++.





Figuur 2.9: Uitvoeringstijd van expression template code (volle lijn) ten opzichte van de versie met iterators (kruisjes) op een Pentium II 266 MHz (boven) en een Pentium 4 2.0 GHz (onder). Horizontaal is de grootte van de vector waarop getest werd uitgezet (n) en vertikaal de uitvoeringstijd in milliseconden (t). Compilatie met GCC g++ Debian versie 3.4.3-12 -O2.

Voordelen

Door het gebruik van expression templates kun je een wiskundige uitdrukking schrijven zonder dat daarbij alle implementatiedetails zichbaar zijn. Die details worden verborgen in de template code die het stukje code dat je schrijft implementeert. Zo krijg je een zeer grote flexibileit qua ontwerp. Het voorstellen van ijle matrices in Fortran is bijvoorbeeld niet eenvoudig. Zeker bewerkingen op ijle matrices zijn niet elegant te schrijven in Fortran. Met expression templates kunnen al de niet relevante details verborgen worden voor de gebruiker. Tegelijkertijd kun je ervoor zorgen dat er een uniforme interface is voor alle type matrices. Als je in Fortran hetzelfde wil, zal je ijle matrices moeten opnemen in de standaard en alle compilers zullen die standaard moeten implementeren.

Nadelen

Expression templates hebben een aantal nadelen

- Ze zijn niet eenvoudig om te programmeren.
- Alle parameters moeten gekend zijn op het moment van compilatie.
- Foutboodschappen bij compilatie zijn door het gebruik van templates soms totaal onverstaanbaar. Dit maakt het schrijven en het gebruik van dergelijke code soms moeilijk.
- Compilatie van de code duurt langer dan bij code die niet uitgebreid templates gebruikt.

Een bijkomend nadeel is, dat de implementatie van dezelfde functionaliteit in de verschillende bestaande expression template libraries niet dezelfde is. Veel functionaliteit zit verspreid over verschillende losse projecten en je kunt niet anders dan de bibliotheek te kiezen die het best datgene ondersteunt, wat je denkt nodig te hebben. Dit houdt het risico in dat later blijkt dat de bibliotheek die je gekozen hebt niet voldoet, en je naar een andere bibliotheek met een verschillende interface moet overstappen. Je zult in dat geval al je code moeten aanpassen aan de nieuwe interface met alle problemen van dien.

2.5 Conclusie

C++ laat toe twee programmeerstijlen te combineren. Langs één kant kun je alle mogelijkheden die C++ voor objectgericht ontwerp biedt gebruiken voor de delen van het programma die geen kritieke code zijn. Langs de andere kant kun je een beperkte deelverzameling van de objectgerichte mogelijkheden van C++ gebruiken om kritieke code te schrijven.

Om kritieke code in C++ echt efficiënt te laten lopen, is een goede kennis van de mogelijkheden die C++ biedt om code te optimaliseren onontbeerlijk. Een belangrijk principe is dat je beslissingen die normaal tijdens de uitvoering van het programma genomen worden zoveel mogelijk tijdens de compilatie laat nemen. We denken hierbij in de eerste plaats aan template-technieken die run time polymorfisme omvormen in compile time polymorfisme.

Je moet nogal wat dingen in het achterhoofd houden bij het programmeren van kritieke code in C++. Vooral het gebruik van pointers in C++ zorgt bij beginners voor veel problemen. Pointers slecht of niet gebruiken kan in C++ een grote impact op de snelheid hebben.

De vele extra mogelijkheden die C++ biedt, laten toe om voor de meeste efficiëntieproblemen een geschikte oplossing te vinden. Samenvattend kun je volgende richtlijnen geven:

- zoveel mogelijk statische in plaats van dynamische keuzes gebruiken door middel van template-technieken

- geen grote klassenhiërarchiën gebruiken
- niet onnodig delegeren van verantwoordelijkheden door middel van encapsulatie
- oordeelkundig gebruik van inlining en pointers

C++ laat je toe om de mogelijkheden van de taal constant uit te breiden zonder dat hiervoor de standaard aangepast moet worden. We denken hierbij in de eerste plaats aan expression templates die complexe implementaties achter bewerkingen volledig voor de gebruiker kunnen verbergen. Zo kun je dichter bij de wiskundige uitdrukking schrijven en toch efficiënte code hebben.

Eén nadeel is dat standaardisatie van deze nieuwe mogelijkheden langer op zich laat wachten. Wij hopen in het belang van numeriek ontwerp in C++ dat er binnen afzienbare tijd een standaard bibliotheek zal komen die een vaste interface voor lineaire algebra operaties vastlegt. Momenteel ben je verplicht, ofwel externe aanroepen van BLAS en LAPACK code te doen, ofwel één van de vele bestaande C++ bibliotheken te kiezen. De eerste oplossing is bijzonder onelegant en een bron van programmeerfouten. De tweede oplossing heeft het nadeel dat je moet kiezen en pas na ervaring met de bibliotheek in kwestie weet of ze voldoet. Als dan uiteindelijk blijkt dat de bibliotheek niet voldoet, kan je proberen over te stappen op een andere bibliotheek. Die andere bibliotheek heeft meestal een andere interface en het resultaat is, dat je de code die je reeds hebt geschreven volledig moet aanpassen. Als er een vaste interface zou bestaan, kan iedereen zich houden aan die ene interface voor het schrijven van programma's en hoeven verschillende implementaties van die interface niet meer tot programmeerproblemen te leiden. De aandacht en energie zal ook meer geconcentreerd zijn op één project met dezelfde idee, daar waar die nu verspreid is over verschillende projecten.

Een bemoedigend project in die richting is GLAS (Knapen [2005]) dat ook deel uitmaakt van het Boost-project (Boost.org [2005]). Een aantal van de Boost-bibliotheken zijn opgenomen in het library technical report (C++ Standard Committee [2005]) van het C++ standaard comité. Dit is een eerste stap op de weg naar standaardisatie. Momenteel probeert men in het GLAS-project door middel van discussie een standaard interface uit te werken voor lineaire algebra operaties in C++.

Het feit dat Fortran 2003 mogelijkheden, die vroeger alleen in C++ te vinden waren, overneemt zien we niet direct als een bedreiging voor C++. De leveranciers van Fortran compilers staan momenteel niet echt te springen om bijvoorbeeld de objectgerichte mogelijkheden die de nieuwe standaard vraagt in hun compilers op te nemen.

Samenvattend is het schrijven van efficiënte code niet eenvoudig in C++. Daar staat tegenover dat C++ toelaat om de mogelijkheden van taal uit te breiden en efficiënte code eenvoudig schrijfbaar te maken.

2.6 Zin en onzin van technisch wetenschappelijk ontwerp in Java

Java had vanaf zijn introductie aantrekkingskracht op ontwikkelaars van numerieke code. In de eerste plaats lijkt Java syntaxtisch sterk op C++ code. Daarenboven heeft java de volgende voordelen ten opzichte van C++:

- standaard klassenbibliotheken met natuurlijke interfaces voor onder andere multithreading, netwerkcommunicatie en protocols, een aanpasbaar veiligheidsmodel, grafische componenten en GUI's

- overdraagbaarheid van code over alle platformen waarop Java draait
- geen gebruik van pointers
- eenvoudiger en automatisch geheugenbeheer door middel van garbage collection
- geen meervoudige overerving en operator overloading wat soms tot zeer onduidelijke code leidde
- Java programma's kunnen *veilig* uitgevoerd worden in een *sandbox* eventueel als applet
- vanaf de laatste versie 1.5 is er ondersteuning voor *generics*

Toch blijkt dat numerieke toepassingen in Java zeer schaars zijn en is het enthousiasme danig bekoeld. In deze sectie zetten we op een rij waarom.

2.6.1 Snelheid van de code

Ondanks het gebruik van Just In Time technieken in de JVM blijft de snelheid van Java code ten opzichte van vergelijkbare C++ of Fortran code in veel gevallen lager. Daar zijn een aantal goede redenen voor.

Excepties

Java verplicht er zich toe om elke referentie te controleren op **null**-referenties en in dat geval een exceptie te gooien. Daarnaast controleert Java voor elke referentie naar een element uit een array of die referentie binnen de grenzen van de array valt. Indien niet moet een exceptie gegooid worden.

Dit is zeer handig tijdens de ontwikkeling van code. Eens gedebugged blijft de code vertraagd worden hierdoor. Uit testen (Pozo [1997]) blijkt dat met die testen op hun plaats en gebruik van de mogelijkheden van de huidige processoren je nog altijd een overhead van 20% hebt. Zelfs al gebruik je optimaal alle mogelijkheden die de huidige architecturen hebben, je blijft met een serieuze vertraging zitten. Los daarvan kunnen die mogelijkheden maar half benut worden door het ontbreken van een matrix type in Java.

Arrays en matrices

Matrices moeten in Java voorgesteld worden als arrays van arrays. Dit betekent dat analyse van de code nodig is om na te gaan of een array van arrays wel degelijk een matrix is of niet. Dit is op zich al een complex analyseprobleem voor een compiler. Dit analyseprobleem slaagt meestal niet. Dergelijke analyse gebeurt niet in de huidige Java compilers. Bijgevolg kan eenvoudige optimalisatie van matrix code met loop transformaties niet gebruikt worden.

Het voorstellen van matrices als arrays van arrays schept nog een tweede probleem: aliasing (zie sectie 2.2.2 op pagina 21). Het kan zijn dat stukken van een array, die misschien een matrix zou kunnen zijn, naar dezelfde array wijzen of dat stukken van twee potentiële matrices gemeenschappelijk zijn. Er is vooralsnog geen manier, zoals in C++, om de compiler duidelijk te maken dat dit niet het geval is voor een bepaalde array in Java. Hierdoor kunnen nog een aantal belangrijke matrixoptimalisaties niet uitgevoerd worden in Java.

Gezien matrices en vectoren niet echt door Java ondersteund worden, kan Java niet efficiënt onderdelen uit een vector of matrix halen.

In numerieke code zijn BLAS-operaties, die fundamenteel op matrices werken, zeer belangrijk. Dit schept voor numerieke code in Java een groot efficiëntieprobleem. De optimalisaties kunnen ook handmatig geschreven worden in de code maar dit veronderstelt een grondige

kennis van de machine en de Java Virtual Machine waarop de code wordt uitgevoerd. De code zal hierdoor minder overdraagbaar worden, omdat ze op de ene machine zeer goed zal werken en op een andere veel minder goed. Eén van de belangrijkste redenen om Java te gebruiken is, dat de code vlot overdraagbaar is en op alle platformen minstens even goed werkt.

Complexe getallen

Java ondersteunt geen complexe getallen. Die kunnen wel gedefinieerd worden in een nieuwe klasse maar omdat Java geen operator overloading heeft kan je die gedefinieerde complexe getallen niet hanteren zoals bij gewone doubles of integers (zie sectie 2.6.2 op pagina 37). Die definitie is veel minder efficiënt dan ze in andere talen is ten opzichte van floating point getallen met dezelfde precisie. De Java compiler kan uit een klassedefinitie de optimalisaties die op code met complexe getallen kan toegepast worden niet afleiden. Daarnaast zorgt het werken met objecten voor meer overhead dan het werken met types eigen aan de taal.

IEEE 754

De huidige Java-specificatie laat ondertussen toe dat dezelfde numerieke operatie niet op alle platformen hetzelfde resultaat moet geven (Boisvert et al. [1996]). Je kan dit wel opleggen met het sleutelwoord **strictfp**. Dit laat alvast toe in een aantal gevallen beter de hardware ondersteuning voor floating point getallen op een platform te gebruiken. Desondanks zijn heel wat optimalisatiemethodes voor floating point bewerkingen nog altijd verboden omdat Java toch een zo gelijk mogelijk resultaat over alle platformen wil garanderen. Bijvoorbeeld is het reduceren van de operatie $ax + y$ tot één hardware instructie (FMA[7]) verboden onder de huidige Java specificatie. Dit is een grote handicap voor de snelheid van numerieke software.

2.6.2 Eenvoud in programmeren

Operator overloading

Operator overloading is weggelaten uit Java omdat dit in andere talen soms tot code leidde die totaal onbegrijpbaar is. Voor numeriek ontwerp is dit een groot nadeel. Zelfgedefinieerde types kunnen zo niet eenvoudig gebruikt worden. Een aantal eenvoudige bewerkingen op complexe getallen ziet er als volgt uit

```
b = a.multiply(c).plus(d);
```

in plaats van

```
b = (a * c) + d;
```

Los van het feit dat het de code minder leesbaar maakt, is het ook niet handig om te schrijven en is de kans op typfouten veel groter. Voor een eenvoudig voorbeeld gaat dat nog, maar als je bijvoorbeeld conjugate gradients implementeert, krijg je figuur 2.10a op pagina 38 terwijl dat er als figuur 2.10b zou kunnen uitzien.

Het weglaten van operator overloading maakt de Java standaard eenvoudiger en niet-numerieke code meer leesbaar. Bij numerieke toepassingen is het een handicap omdat je regelmatig zelfgedefinieerde structuren nodig hebt. Operaties op die zelfgedefinieerde structuren kunnen met operator overloading meer leesbaar gemaakt worden.

[7] Fused Multiply Add

Figuur 2.10: Verschil tussen code met operaties op complexe getallen zonder en met operator overloading.

(a) Zonder operator overloading

```
d = r = b.minus(A.multiply(x));
while( ... )
{
    a = r.transpose().multiply(r).divide(d.transpose().multiply(A).multiply(d));
    x = x.plus(a.multiply(d));
    r_old = r;
    r = r.minus(a.multiply(A).multiply(d));
    beta =
    r.transpose().multiply(r).divide(r_old.transpose().multiply(r_old));
    d = r.plus(beta.multiply(d));
}
```

(b) Met operator overloading

```
d = r = b - A*x;
while ( ... )
{
    a = (r.transpose()*r) / (d.transpose() * A * d);
    x = x + a*d;
    r_old = r;
    r = r - (a*A*d);
    beta = (r.transpose() * r) / (r_old.transpose() * r_old);
    d = r + beta*d;
}
```

IEEE 754

Java ondersteunt, net als de meeste talen, niet alles wat de IEEE 754 standaard specifieert. De afrondingsmodus is niet onder de controle van de programmeur en floating point excepties kunnen niet gegooid worden. Dit laatste zou het debuggen en schrijven van numerieke toepassingen aanzienlijk kunnen vereenvoudigen.

2.6.3 Eventuele oplossingen

Java Native Interface

Het probleem van de snelheid van Java code en libraries kan opgelost worden door gebruik van libraries die geschreven zijn voor het platform waarop het Java programma wordt uitgevoerd aan de hand van de Java Native Interface (JNI). Gebruik van JNI schept een aantal problemen (Boisvert et al. [1996]):

- Overdraagbaarheid gaat verloren
- "Absolute" veiligheid gaat verloren
- Minder robuste code
- Indien kleinere hoeveelheden informatie doorgegeven moet worden door de JNI en die JNI veelvuldig aangeroepen wordt, gaat er veel snelheid verloren in het doorgeven van de informatie naar de native library.
- Het resultaat kan verschillend zijn afhankelijk van het platform

Een aantal van deze problemen bestaan ook met code in C++ maar die problemen zijn er niet als we de JNI niet gebruiken, en dit is juist één van de redenen om Java te gebruiken. Die problemen kunnen gedeeltelijk opgelost worden door automatische vertaling van bestaande goed werkende libraries naar Java, maar dit lost het snelheidsprobleem niet op. In het NetSolve project (Casanova et al. [1997]) gebruikt men de netwerkmogelijkheden van Java om parallelle uitvoering van Java code over verschillende computers te spreiden en bepaalde operaties *native* uit te voeren aan de hand van JNI.

Operator Overloading

Het ziet er niet naar uit dat men ooit operator overloading aan Java zal toevoegen. Er zijn zeer goede redenen waarom Java *geen* operator overloading heeft. Operator overloading kan misbruikt worden om vanzelfsprekende operatoren een tegenintuïtieve werking te geven. Het resultaat is code die niet meer te begrijpen is (Javagrande [1998]). In numerieke code is dit meestal niet het geval en verhoogt dit juist de leesbaarheid en schrijfgemak van de code indien het niet misbruikt wordt.

Afzetten van array bounds checking

Dit zou de snelheid van code, die grondig getest is en waarvan je vrij zeker bent dat ze geen fouten zal begaan tegen het overschrijden van de grenzen van arrays, kunnen verbeteren. Het is zeer twijfelachtig of men dit zal doen. Ten eerste gaat het grondig in tegen het fundamentele veiligheidsconcept van Java. Ten tweede kan dit probleem gedeeltelijk aangepakt worden met gebruik van hardwarefuncties op moderne processoren (Moreira et al. [2000]).

2.6.4 Conclusie

Uit een aantal tests (Boisvert et al. [1996] en SciMark (Pozo and Miller [2004]) blijkt dat in bepaalde gevallen de snelheid van Java in de buurt kan komen van C en C++ . Het gaat hier over een zeer specifieke set testen. Het is maar zeer de vraag of dit de werkelijkheid goed benadert.

Er zijn een tijdlang zeer veel initiatieven van het Java Grande Forum gekomen (zie JavaGrande [2003] en Pozo and Boisvert [2005]). De mensen van dat forum hebben een aantal voorstellen gedaan om Java uit te breiden of aan te passen voor numerieke software. Dit forum is momenteel zo goed als uitgestorven.

Uit een opmerking van Ronald F. Boisvert, één van de belangrijkste personen achter het Java Grande Forum, op een conferentie in Gent in 2004 blijkt ook, dat zij momenteel niet meer actief bezig zijn met lobbyen bij Sun. De reden blijkt niet te zijn dat Sun niet op hun ideeën wilde ingaan. Dit bleek uit een mail die ik van Boisvert ontving:

> From: Ron Boisvert
> Subject: Re: Java, Numerics and Sun
>
> The Java Grande effort has indeed gone dormant. All of the "easy" things to improve Java had been done. Further improvements would have required sheparding proposals through the Java Community Process for standardization, along with a committment of long-term support. None of the participants had the patience, or time, to do this.
>
> Sun has always been cooperative, and would probably be happy to have continued interest from the scientific computing community.
>
> Some of the enthusiasm also waned when Microsoft's C# system was unveiled with many of the language features that we had been lobbying for. The disappointing thing is that C# performance using these features remains sluggish at best.
>
> At NIST we continue to maintain the JavaNumerics Web page, and to support the SciMark benchmark.
>
> Best wishes, Ron Boisvert

Tot iemand de draad terug opneemt zal in deze situatie geen verandering komen.

Hoofdstuk 3

Functionele talen en numerieke toepassingen

Als je een blik werpt op het merendeel van de wiskundige algoritmes, dan zie je dat ze bijna zonder uitzondering procedureel gestructureerd zijn. Dit verklaart waarom een taal als Fortran, die tot voor versie 2003 een procedurele taal was, zoveel aanhang heeft bij ontwerpers van numerieke toepassingen.

Een andere reden is dat een wetenschapper, die op het moment dat hij voor het eerst een programmeertaal moet kiezen, in de eerste plaats naar een taal zal grijpen die aansluit bij het denken dat hij gebruikt om algoritmes te formuleren.

Men kan hier eigenlijk spreken van een paradigma. Hoewel puur wiskundige formuleringen meestal niet procedureel gestructureerd zijn, wordt, voor de overgang van de wiskundige formulering en oplossing naar een praktische implementatie ervan, doorgaans gebruik gemaakt van een procedurele structurering van het algoritme. Dit is een gegeven dat zichzelf voedt en op de duur lijkt het of elk algoritme procedureel *moet* gestructureerd zijn. Dat een algoritme *niet* noodzakelijk procedureel geformuleerd en geïmplementeerd moet worden om praktisch bruikbaar te zijn, zal blijken in dit hoofdstuk. Daarnaast zijn we op zoek naar een taal, die qua concepten zo dicht mogelijk bij de wiskundige formulering van een probleem ligt.

Details aangaande het schrijven van programma's in Haskell, de functionele taal die wij gekozen hebben, kan je in de desbetreffende standaardwerken vinden (Thompson [1999] en Jones [2002]). Een lezer die niet bekend is met Haskell zal merken dat het schrijven van programma's in deze taal niet zo moeilijk is. Het design van de functionele talen die voor Haskell kwamen had heel wat invloed op dat van Haskell. Voor meer details hierover zie Hudak [1989].

3.1 Mogelijkheden

Het belangrijkste onderscheid tussen functionele talen en andere talen is dat het oplossen van een probleem louter en alleen gebeurt door het evalueren van uitdrukkingen. Daarom is geen strikte volgorde opgelegd aan welke uitdrukking eerst geevalueerd wordt. Soms wil je wel een strikte ordening en de meeste functionele talen laten toe om indien nodig van deze programmeerstijl af te wijken. In de regel heb je dit niet nodig behalve voor IO.

3.1.1 Analogie met de wiskunde

Functies worden in de wiskunde gedefinieerd aan de hand van een voorschrift waarin eventueel één of meerdere parameters gebruikt worden. Dit is ook zo in Haskell

- Wiskunde: $f(x) = 1 - x^2$
- Haskell:

```
f x = 1 - x^2
```

In de wiskunde zal je meestal ook het domein waarover de functie gaat opgeven. Dit kun je ook in Haskell

- Wiskunde: $f : \mathbb{R} \to \mathbb{R} : (x) \mapsto 3 - x^2$
- Haskell:

```
f :: Double -> Double
f x = 3 - x^2
```

Merk op dat het datatype **Double** niet gelijk is aan $\mathbb{R}$.

In nogal wat wiskundige definities kan je het domein weglaten. De definitie zal blijken op te gaan over zeer veel domeinen. Ook in Haskell kan je in veel gevallen de domeinspecificatie weglaten. In Haskell heet de domeinspecificatie van een functie de typespecificatie of signatuur. Als die niet gegeven is probeert Haskell die af te leiden aan de hand van *type inference*. Meer over de implicaties hiervan vindt u in sectie 3.2.4 op pagina 53.

3.1.2 Functies

Het concept van functies is in functionele talen anders dan in niet functionele talen. Een functie heeft in een functionele taal altijd hetzelfde resultaat als het exact dezelfde argumenten krijgt. Dit is ook zo in de wiskundige formulering van een functie. Een dergelijke functie is een pure functie. Een uitzondering hierop in Haskell zijn *monads* die gebruikt worden voor het geval je niet-pure functies wilt schrijven, bijvoorbeeld voor IO.

Een functie met meerdere argumenten krijgt volgende signatuur

```
add :: Int -> Int -> Int
add x y = x + y
```

De notatie die gebruikt wordt op de eerste lijn zal duidelijker worden als we het over currying hebben. Deze functie heeft twee argumenten van het type **Int** en geeft een resultaat van het type **Int** terug.

In Haskell kan je net zoals in de meeste functionele talen lambdafuncties schrijven. De lambdafunctie voor $f = x^2$ is in Haskell **(\x -> x^2)**. Hiermee kun je snel eenvoudige functies als argument meegeven aan een andere functie zonder dat je die eerst afzonderlijk moet definiëren en een unieke naam geven.

Haskell laat vanzelfsprekend toe om functies te nesten anders zou je teveel losse functies moeten schrijven en teveel argumenten moeten doorgeven. Dit kan op twee manieren. Met **where** of met **let ... in**. Door *type inference* moet je bij geneste functies zelden of nooit een typespecificatie opgeven. Na **where** definieer je functies die je voor **where** gebruikt. Op dat niveau kunnen functies ook naar mekaar verwijzen. **let** werkt omgekeerd. Na **let** definieer

Figuur 3.1: Conjugate gradients in Haskell

```
cg b a x = iterate (conjug) (d,r,x)
  where
    d = r
    r = b - a<*>x
    conjug (d,r,x) = (d_new,r_new,x_new)
      where
        d_new = r_new + beta <*> d
        r_new = r - (a_<*>a<*>d)
        x_new = x+ a_<*>d
        a_ = ((transpose r) <*> r) / ((transpose d) <*> a <*> d)
        beta = ((transpose r_new) <*> r_new ) / ((transpose r) <*> r)
```

je functies die je na **in** gebruikt. Zo haal je eenvoudig complexe uitdrukkingen uiteen in hun logische componenten zonder dat je types moet specifiëren voor elk onderdeel. Op die manier zou conjugate gradients er als figuur 3.1 op pagina 43 kunnen uitzien. De functie **iterate** itereert de functie **conjug** startende vanaf de lijst met de drie startelementen en creëert zo een oneindige lijst. Elk volgend element uit de lijst is een verdere iteratie. Merk op dat de functie **conjug** in **cg** slechts één argument heeft. Deze manier van schrijven verduidelijkt de logische structuur van het algoritme.

3.1.3 Currying en hogere orde functies

Hogere orde functies zijn functies die doorgegeven kunnen worden als argument of teruggegeven kunnen worden als resultaat van een functie. De lezer zal al opgemerkt hebben dat lambda-expressies hogere orde functies zijn. Daarnaast kan elke functie in Haskell als hogere orde functie behandeld worden.

Currying betekent dat niet alleen een functie zonder argument als functietype beschouwd wordt, zoals **conjug** in figuur 3.1, maar ook een functie met een onvolledig aantal argumenten.

Als we het voorbeeld met **add** uit sectie 3.1.2 hernemen dan heeft de functie **add 2 3** het type **Int** en als resultaat **5**. De functie **add 2** is van het type **Int -> Int**. Dit betekent dat het een functie is die één **Int** als argument neemt en als resultaat een **Int** teruggeeft. Je zou **add 2 3** kunnen schrijven als **(add 2) 3**. Dit betekent dat **3** als argument gegeven wordt aan een functie die het getal **2** optelt bij zijn argument. De uitdrukking **add 2** kun je beschouwen als **2** die als argument gegeven wordt aan de functie **add** en die als resultaat een functie teruggeeft die het getal **2** bij een ander getal optelt.

In een niet functionele taal moet je, om hetzelfde effect te bekomen, een nieuwe functie schrijven. In Haskell laat je, in volgorde van achter naar voor, één of meerdere argumenten weg. Doorgeven van functies aan andere functies is een evidentie in Haskell. Als je alle elementen van een lijst wil delen door hetzelfde getal dan schrijf je in Haskell **fmap (/2) lijst**. Met **fmap** pas je een gegeven functie toe op elk element van een lijst. Die functie, in dit geval **(/2)**, kan om het even wat zijn, zolang het maar iets doet met iets van het type van de elementen van de lijst. Dit is een veel veiliger alternatief dan het schrijven van een for-loop met indices.

For-loops bestaan niet in Haskell en worden vervangen door het recursief of itererend aanroepen van een functie. Het resultaat van conjugate gradients uit figuur 3.1 berekenen na de 20ste iteratiestap is het opvragen van het 20ste element uit de lijst met de resultaten. Je

schrijft **(cg b a x)!!20**. **!!** vraagt een element op uit de lijst die ervoor komt.

3.1.4 Pattern matching en guards

Pattern matching en guards zijn onder andere een alternatief voor *case* en *if-then-else*-structuren. Voor een functie die de respectieve elementen van een lijst optelt, hoef je niet noodzakelijk de code als in figuur 3.2a of figuur 3.2b op pagina 45 te schrijven voor het optellen van twee lijsten. Met pattern matching kun je code als in figuur 3.2c schrijven. [] staat hier voor de lege lijst en **(x0:xs)** is pattern matching voor lijsten en betekent dat **x0** refereert naar het eerste element van de lijst en **xs** naar de rest van de lijst. Zonder pattern matching op lijsten zou je functies moeten gebruiken om expliciet het eerste element en de rest van een lijst op te halen (respectievelijk **head** en **tail**) zoals in figuur 3.2d.

Pattern matching zoekt naar patronen in functiedefinities, lijsten, datatypes, . . . Het eerste patroon dat past zal Haskell gebruiken. Bij functies die meerdere keren gedefinieerd zijn zal Haskell de definitie gebruiken waarvan de argumenten het eerst passen. Het verduidelijkt veelal wat bedoeld wordt en het is intuïtiever.

Wanneer de voorwaarden in een uitgebreide *if-then-else*-structuur te complex worden om nog met pattern matching te vereenvoudigen, kun je gebruik maken van guards. Dan ziet figuur 3.2e er als figuur 3.2f uit. Guards zijn sneller geschreven dan *if-then-else*-structuren en ze lezen sneller.

Belangrijk bij pattern matching en guards is de volgorde van de regels. Haskell begint altijd bovenaan. Het spreekt voor zich dat je het meest voorkomende geval bovenaan plaatst, zodat geen onnodige vergelijkingen uitgevoerd moeten worden.

3.1.5 Lazy evaluation

Lazy evaluation is het uitstellen van de evaluatie van het argument van een functie tot op het moment dat het absoluut nodig. Een eenvoudig voorbeeld

```
ones = 1 : ones

first (u0:ux) = u0
```

Als je **first ones** opvraagt wordt niet eerst een oneindige structuur met allemaal enen in opgesteld, maar wordt alleen het eerste element van de functie **ones** opgevraagd. Dit is een zeer handig mechanisme voor het omgaan met onder andere oneindige datastructuren. Sectie 3.3 op pagina 53 komt daar uitgebreid op terug.

3.1.6 Zelfgedefinieerde datatypes

In Haskell is het eenvoudig nieuwe datatypes aan te maken. Een datatype dat quarks modelleert ziet er als figuur 3.3a op pagina 46 uit. Je leest dit als volgt: het datatype **Quark** kan **Blauw**, **Rood** of **Groen** zijn en heeft telkens een waarde die de spin aangeeft. Het teken | staat hier voor *of.* De parameter **spin** in deze definitie geeft aan dat deze datadefinitie nog een typeparameter heeft. **Blauw 1.5** kan[1] van het type **Quark Float** zijn.

Stel dat de botsing tussen twee quarks van een verschillend type een quark van het andere type dan de twee botsende quarks geeft met de som van hun spins, dan kun je dit met pattern matching modelleren zoals in figuur 3.3b. Bemerk dat je het symmetrische geval niet

[1]Voor de redenen waarom we hier *kan* gebruiken zie sectie 3.2.4 op pagina 53

Figuur 3.2: Pattern matching en guards

(a) Optellen van de waarden uit twee lijsten met een *case*-structuur.

```
telop x y = case x of
    [] -> y
    n -> case y of
       [] -> x
       n -> (head x + head y):telop (tail x) (tail y)
```

(b) Optellen van de waarden uit twee lijsten met een *if-then-else*-structuur.

```
telop x y = if (x == []) then y
            else if (y == []) then x
                 else (head x + head y):telop (tail x) (tail y)
```

(c) Optellen van de waarden uit twee lijsten met pattern matching.

```
telop [] y = y
telop x [] = x
telop (x0:xs) (y0:ys) = (x0+y0): telop xs ys
```

(d) Optellen van de waarden uit twee lijsten met pattern matching op functie-argumenten maar zonder pattern matching op lijsten.

```
telop [] y = y
telop x [] = x
telop x y = (head x) + (head y): telop (tail x) (tail y)
```

(e) Stukje code uit de BLAS routine CHBMV zonder guards.

```
info = if ( not lsame(uplo, 'U' ) && not lsame(uplo, 'L' ) )  then 1
       else if (n < 0 ) then 2
       else if ( k < 0 ) then 3
       else if (lda < (k+1)) then 6
       else if (incx == 0) then 8
       else if (incy == 0) then 11
       else 0
```

(f) Hetzelfde stukje code uit de BLAS routine CHBMV met guards.

```
info | not lsame(uplo, 'U' ) && not lsame(uplo, 'L' ) = 1
     | n < 0        = 2
     | k < 0        = 3
     | lda < (k+1)  = 6
     | incx == 0    = 8
     | incy == 0    = 11
     | otherwise    = 0
```

Figuur 3.3: Zelfgedefinieerde datatypes

(a) Een quark data type.

```
data Quark spin => Blauw spin | Rood spin | Groen spin
```

(b) Een functie die iets met een quark doet.

```
interacts (Blauw spin1) (Rood spin2) = Groen (spin1+spin2)
interacts (Blauw spin1) (Groen spin2) = Rood (spin1+spin2)
interacts (Groen spin1) (Rood spin2) = Blauw (spin1+spin2)
interacts x y = interacts y x
```

(c) Definitie van een boom data type met integers in.

```
data Tree = Leaf Int | Node Int Tree Tree
```

(d) Definitie van een boom data type met data van een willekeurig type in.

```
data Tree a = Leaf a | Node a Tree Tree
```

(e) Een functie die alle waarden uit een gegeven boom optelt.

```
sum (Leaf x) = x
sum (Node x y z) = x + (sum y) + (sum z)
```

moet herhalen. Je roept **interacts** nogmaals aan maar met de volgorde van de argumenten omgekeerd. In werkelijkheid zijn de interacties tussen quarks uiteraard veel complexer.

De gedefinieerde datatypes kunnen ook recursief zijn. Een boomstructuur ziet er als figuur 3.3c uit. Een boom is

- ofwel een blad met een integer in: **Leaf Int**
- ofwel een knoop met een integer in en een linker- en rechterboom: **Node Int Tree Tree**

Een functie om alle waarden uit een boom op te tellen gebruikt pattern matching (zie figuur 3.3e). Gecombineerd met het klassensysteem van Haskell is het niet moeilijk om nieuwe types met operaties erop te definiëren. Je kan ook het type waarover een boom gaat abstraheren en dan ziet de datadefinitie er als figuur 3.3d uit. De functie **sum** werkt net zo goed op deze definitie zolang de operatie + maar op het datatype gedefinieerd is.

Het tonen van nieuwe datatypes op het scherm kan ook eenvoudig gebeuren door bij de definitie **deriving Show** op te geven. Haskell zorgt er zelf voor dat het tonen van gegevens van een bepaald datatype geen extra werk kost.

Als men een datatype gebruikt om alle gegevens van een uitgebreid probleem te bevatten, krijg je een zeer lang datatype. Dat datatype bestaat hoofdzakelijk uit een opsomming van parameters en het wordt onhandig om nog met patternmatching te gebruiken. Je kan dan best gebruik maken van veldlabels.

```
data Vlak a = Vlak {xmin,xmax,ymin,ymax:: a, temperatuur:: Double}
```

Figuur 3.4: Overloading

(a) Een typeklasse met twee operaties erop.

```
class Number a where
    (<+>) :: a -> a -> a
    negate :: a -> a
```

(b) Definitie van de operaties <+> en **negate** op **Number**s van het type **Double**.

```
instance Number Double where
    (<+>) = (+)
    negate = (0.0 -)
```

(c) Definitie van de operaties <+> op **Number**s van het type **Quark**.

```
instance Num a => Num (Quark a) where
    x + y = interacts x y
```

(d) De typeklasse **Number** erft de eigenschappen van **Eq**.

```
class Eq a => Number a where
    (<+>) :: a -> a -> a
    negate :: a -> a
```

Het veld **temperatuur** uit een expressie **x** halen dat een **Vlak** is doe je door **temperatuur x** te schrijven. In een expressie **x** de waarde voor **temperatuur** aanpassen doe je door **x {temperatuur = 0.1}** te schrijven.

3.1.7 Overloading en parametrisch polymorfisme

Overloading valt te onderscheiden van parametrisch polymorfisme. Om met beide concepten om te gaan heeft men in Haskell de notie van typeklassen geïntroduceerd. Parametrisch polymorfisme betekent, dat het voor een functie niet uitmaakt welk type het argument heeft, zolang er maar een aantal operaties of functies op dat type gedefinieerd zijn. Men geeft aan dat een datatype tot een bepaalde typeklasse behoort, als op dat datatype een aantal operaties en/of functies gedefinieerd zijn met een bepaalde signatuur.

Men zou kunnen zeggen dat een datatype bij de klasse van de nummers hoort, als de operatie + en de negatie erop gedefinieerd zijn. In figuur 3.4a staat **a** voor een willekeurig datatype. Je kunt vervolgens functies schrijven die als signatuur **Number a => a -> a** hebben en gebruik maken van operaties die tot de typeklasse **Number** behoren. Een dergelijke functie maakt gebruik van parametrisch polymorfisme. Zolang het argument van die functie tot de klasse **Number** behoort, zal er een resultaat zijn. De notatie **Number a =>** legt deze beperking op aan de signatuur van deze functie.

Bij ad hoc polymorfisme of overloading gaat het er juist om die operaties of functies, in dit geval <+> en **negate**, voor een bepaald datatype te specifiëren. Dat heet overloading. Als je het Haskell datatype **Double** ook wilt kunnen bewerken met <+> en **negate** dan kun je code als in figuur 3.4b schrijven. Dit wordt pas echt nuttig als je het over eigen datatypes gebruikt. Stel dat je quarks kunt optellen dan gebeurt het overloaden over quarks met een

instantiedeclaratie zoals in figuur 3.4c. Telkens je twee variabelen van het datatype **Quark** optelt met + wordt de gegeven functie **interacts** gebruikt om de twee variabelen op te tellen. Gezien de operator + tot de klasse **Num** behoort, moet je een instantiedeclaratie van de klasse **Num** geven.

3.1.8 Overerving

Het concept van overerving is verschillend van dat van objectgerichte talen. Overerving gaat alleen van de ene typeklasse naar de andere. Dit heeft als gevolg dat een instantiedeclaratie waarbij een datatype deel gemaakt wordt van een typeklasse onderaan een overervingsboom moet vergezeld worden door instantiedeclaraties waarbij dat datatype deel gemaakt wordt van alle typeklassen waarvan de onderste typeklasse erft.

Als je **Number** wilt laten overerven van de standaard Haskell klasse **Eq**, dan verandert de definitie in figuur 3.4a in die in figuur 3.4d. De klasse **Eq** is de klasse van alle datatypes waarop de gelijkheidsoperator == is gedefinieerd. De nieuwe definitie van **Number** lees je als "De **Number**s met type **a** moeten ook tot de klasse **Eq**[2] behoren en hebben volgende operaties..."Een datatype dat tot de klasse **Number** behoort, zal ook tot de klasse **Eq** moeten behoren. Een instantiedeclaratie die van **Quark** een **Number**-type maakt, moet altijd vergezeld zijn door een instantiedeclaratie die van **Quark** een **Eq**-type maakt.

Meer aangaande het typesysteem vindt u in sectie 3.2 op pagina 48.

3.1.9 Operatoren

In Haskell kun je zoveel nieuwe operatoren maken als je wil. Elke willekeurige combinatie van de tekens

```
!  #  $  %  &  *  +  .  /  <  =  >  ? @ \ ^  -  ~ |
```

die niet reeds een andere betekenis heeft in Haskell kan een operator zijn. Dit laat toe om naar willekeur nieuwe operatoren te gebruiken. Door het strikte typesysteem is duidelijker wat een operator doet al zien de nieuwe operatoren er niet altijd zo esthetisch uit als je zou willen.

Je kunt ook de sterkte van de associativiteit aangeven voor elke operator. Dit doe je met **infixl** (links), **infixr** (rechts) of **infix** (geen associativiteit). De associativiteitssterkte gaat van 0 tot 9. Als voor een nieuwe operator geen associativiteit opgegeven is dan krijgt die automatisch sterkte 10. Operatoren met eenzelfde sterkte kunnen niet in dezelfde uitdrukking gebruikt worden zonder dat de associativiteit met het gebruik van haken duidelijk gemaakt wordt. Als je plant om nieuwe operatoren te mengen met het gebruik van de standaard Haskell operatoren hou je best rekening met de associativiteitssterkten die in de prelude gedefinieerd zijn voor die operatoren. Dit systeem geeft een grote flexibiliteit bij het definiëren en gebruik van nieuwe operatoren.

3.2 Typesysteem

Deze sectie illustreert, onder andere aan de hand van een aantal voorbeelden uit Fokker [1995], hoe het typesysteem van Haskell werkt. Jeroen Fokker gebruikt Haskell om groepentheorie te

[2] Afkomstig van equality in het engels.

modelleren en illustreren in code. Ook Jerczy Karczmarczuk (Karczmarczuk [1995]) gebruikt Haskell om wiskundige concepten te illustreren.

3.2.1 Groepentheorie

Zowat de eenvoudigste algebraïsche structuur die je je kunt indenken is een monoïde. Een monoïde is een verzameling die gesloten is met een associatieve binaire bewerking en het heeft een neutraal element. Met een klassedefinitie wordt dit figuur 3.5a op pagina 50. De wetten, de associativiteit en die voor een bewerking met een neutraal element, waaraan de operaties moeten voldoen, kun je niet eenvoudig modelleren in Haskell en in de meeste functionele talen kan dit niet. Je kunt geen regels opleggen waaraan een operatie moet voldoen, tenzij je die expliciet in de implementatie van de operatie zelf schrijft. In wat volgt zullen we deze wetten verder niet meer beschouwen of vermelden.

Omdat in de wetten die voor een monoïde gelden de notie van de gelijkheid wordt gebruikt, leid je **Monoid** af van de klasse **Eq**. Dit betekent dat alle instanties van de klasse van de monoïden ook instanties van de klasse **Eq** moeten zijn. Van **Eq** moet alleen de operator == gedefinieerd worden.

Als het ingebouwde type **Int** een monoïde moet zijn, geef je een instantiedeclaratie zoals in figuur 3.5b. In een dergelijke instantiedeclaratie moet je aangeven hoe voor een gegeven type de operaties erop werken. Je kunt bijvoorbeeld `2 <+> 3 <+> zero` schrijven en dan zal het resultaat **5** zijn. Omdat het ingebouwde type **Int** al een instantie is van **Eq**, moet je geen instantiedeclaratie voor **Int** van **Eq** meer geven.

De volgende klasse die je kunt afleiden van een monoïde is een groep (zie figuur 3.5c). Bij een groep moet de omgekeerde onder de optelling gegeven zijn. Dit is de negatie. Als je het datatype **Int** nu ook een groep wilt laten zijn, zul je de declaratie zoals in figuur 3.5d geven. **negate** is de negatie die over **Int** in de prelude gedefinieerd is.

Een ring heeft nog een extra operator en een eenheidselement (zie figuur 3.5e). In een veld heeft elk element behalve nul een inverse (zie figuur 3.5f).

Een euclidisch domein is een ring waarbij een bewerking met deling gedefinieerd is. De rest bij deling moet kleiner zijn dan het deeltal. De relatieve grootte van een getal in een euclidisch domein wordt in onze definitie gesymboliseerd door **norm**. Uit dit voorbeeld blijkt, dat je bij een typeklasse niet beperkt bent tot het geven van bewerkingen, functies en constanten. Je kunt die bewerkingen, functies en constanten ook vastleggen in functie van bewerkingen, functies en constanten die in die typeklasse voor handen zijn. Ook die uit typeklassen, waar de huidige typeklasse van erft, kunnen gebruikt worden.

In dit geval kan de modulobewerking gedefinieerd worden in functie van gelijkheid (overgeërfd van **Eq**) , de optelling en vermenigvuldiging (`<+>` en `<*>` overgeërfd van **Monoid** en **Ring**) en de functie **divide** die in deze typeklasse gedefinieerd is. De laatste lijn van figuur 3.5g kun je lezen als "de definitie van x modulo y is gegeven door de op = volgende definitie als y niet gelijk is aan nul." Deze specificatie van een operatie of functie in een typeklasse is niet bindend. Elke instantiedeclaratie kan ze herdefiniëren.

3.2.2 Toepassingen

Je kunt functies schrijven die over elke monoïde, groep, ring of euclidisch domein geldig zijn. Hierbij geef je telkens de typeklasse op waartoe de argumenten moeten behoren. Het typesysteem met behulp van typeklassen kun je zien als een systeem dat werkt met beperkingen.

Figuur 3.5: Groepentheorie

(a) Klassedefinitie van een monoïde.

```
class Eq a => Monoid a where
    (<+>)   :: a -> a -> a
    zero    :: a
```

(b) Instantiedeclaratie om **Int** een monoïde te laten zijn.

```
instance Monoid Int where
    zero = 0
    (<+>) = (+)
```

(c) Klassedefinitie van een groep.

```
class Monoid a => Group a where
    neg :: a -> a
```

(d) Instantiedeclaratie om **Int** een groep te laten zijn.

```
instance Group Int where
    neg = negate
```

(e) Klassedefinitie van een ring.

```
class Group a => Ring a where
    (<*>) :: a -> a -> a
    one :: a
```

(f) Klassedefinitie van een veld.

```
class Ring a => Field a where
    inv :: a -> a
```

(g) Klassedefinitie van een euclidisch domein.

```
class Ring a => Euclid a where
    norm :: a -> Int
    divide :: a -> a -> a
    modulo :: a -> a -> a
    x 'modulo' y | y /= zero = x <+> neg ((x `divide` y) <*> y)
```

Figuur 3.6: Toepassingen

(a) Algoritme van Euclides in een euclidisch domein.

```
ggd :: Euclid a => a -> a -> a
ggd x y | norm y <= 0 = x
        | otherwise   = ggd y (x `modulo` y)
```

(b) Definitie van gelijkheid en ordening op natuurlijke getallen.

```
instance Eq Nat where
    Zer   == Zer   = True
    Suc x == Suc y = x == y
    _     == _     = False
instance Ord Nat where
    Zer   <= _     = True
    Suc _ <= Zer   = False
    Suc x <= Suc y = x <= y
```

(c) De natuurlijke getallen als monoïde.

```
instance Monoid Nat where
    zero = Zer
    Zer <+> y = y
    Suc x <+> y = Suc (x <+> y)
```

(d) Aftrekking over de natuurlijke getallen.

```
natMin :: Nat -> Nat -> Maybe Nat
natMin x Zer = Yes x
natMin Zer _ = No
natMin (Suc x) (Suc y) = natMin x y
```

(e) Vermenigvuldiging over de natuurlijke getallen.

```
natMul Zer     _ = Zer
natMul (Suc x) y = y <+> natMul x y
```

(f) Deling over de natuurlijke getallen.

```
natDiv x y = case natMin x y
        of No    -> Zer
           Yes d -> Suc (natDiv d y)
```

Het legt een beperking op aan de parameters van een functie en garandeert dat er dan een resultaat is. Als de types van de parameters van een functie niet aan de opgelegde beperkingen in de signatuur voldoen, zal het programma niet compileren. Dit systeem laat toe om elk datatype te doen werken met operaties of functies van een bepaalde klasse, of met functies die over die klasse gedefinieerd zijn. Het enige wat hiervoor nodig is, is een instantiedefinitie die dat datatype een instantie van de klasse maakt.

Het algoritme van Euclides voor de grootste gemene deler werkt over elk euclidisch domein (zie figuur 3.6a op pagina 51). De typespecificatie van **ggd** geeft aan, dat zowel de twee elementen als het resultaat van deze functie moeten behoren tot hetzelfde datatype dat instantie gemaakt is van de typeklasse **Euclid**. Deze typespecificatie legt één beperking op het type van de argumenten en het resultaat van deze functie. Meerdere beperkingen kunnen opgelegd worden.

3.2.3 Natuurlijke getallen

De natuurlijke getallen kun je in Haskell voorstellen met de Peano axioma's. Elk natuurlijk getal is een opvolger van het vorige natuurlijke getal of van nul. Dit druk je uit met de volgende datadefinitie

```
data Nat = Zer | Suc Nat
```

Het getal 2 kun je voorstellen als de tweede opvolger van nul: **Suc (Suc Zer)**.

Om een getal uit deze voorstelling om te zetten naar een integer kun je volgende recursieve functie gebruiken

```
toInt Zer = 0
toInt Suc x = 1 + (toInt x)
```

Gelijkheid en ordening van de natuurlijke getallen wordt in figuur 3.6b gedefinieerd. Merk op dat de tweede definitie bij **Eq** en de laatste bij **Ord** recursief werken. De verzameling van de natuurlijke getallen is ook een monoïde (zie figuur 3.6c). De aftrekking is een partiële functie over de natuurlijke getallen. Er is ofwel een natuurlijk getal als resultaat ofwel geen resultaat. Een dergelijk resultaat definieer je als een Haskell datatype **Maybe**

```
data Maybe a = No | Yes a
```

Als het resultaat negatief is geef je **No** terug. Als het resultaat niet negatief is geef je **Yes resultaat** terug.

De functie **natMin** (zie figuur 3.6d) schrijdt stelselmatig terug tot één van de argumenten **Zer** is. Als het rechterargument nul is, is het resultaat gelijk aan het linker. Anders is er geen resultaat.

Een totale vermenigvuldiging (zie figuur 3.6e) en deling (zie figuur 3.6f) kun je wel opstellen maar dit maakt de natuurlijke getallen geen ring omdat het geen groep is. Voor de deling maak je gebruik van de partiële aftrekking. Je trekt telkens het tweede argument af van het eerste tot je geen resultaat krijgt.

Een dergelijke implementatie is niet moeilijk en je kunt er direct mee experimenteren. Het maakt de wiskundige ideeën achter de Peano-axioma's ook veel duidelijker. Een dergelijke implementatie ondernemen zonder een typesysteem zoals in Haskell vereist andere concepten, zoals bijvoorbeeld klassen en templates, waarmee je dit systeem alleen kunt simuleren en zeker niet in de buurt komen van de eenvoud en de logica van dit systeem voor een dergelijke toepassing.

De juist gedefinieerde structuren zal je niet gebruiken voor numerieke toepassingen die zo efficiënt mogelijk moeten zijn. Met dat doel zijn deze structuren in dit geval niet ontworpen. De bedoeling is dat studenten uit een wetenschappelijke richting abstracties concreter kunnen benaderen door er actief mee te experimenteren. Daarnaast kun je op een snelle manier problemen aanpakken die in een imperatieve taal heel veel extra implementatiedetails vergen die in Haskell niet nodig zijn.

3.2.4 Type inference

Met type inference wordt het werk van de programmeur vereenvoudigd. Waar je in imperatieve talen verplicht bent telkens het type op te geven (of dat nu geparametriseerd is of niet) kun je in functionele talen de typespecificatie overslaan en de afleiding hiervan aan de compiler overlaten. Gezien zo goed als alle onderdelen van een Haskell-programma uit al of niet geneste functies bestaan, is dit geen overbodige luxe. Altijd en overal het type moeten opgeven zou veel van de voordelen van een functionele taal teniet doen. Dit systeem is zeer goed en leidt in heel veel gevallen automatisch af wat de types van alle functies zijn.

Een aantal dingen kan type inference niet. Als je in Haskell code **1** of **1.5** schrijft dan zijn die niet automatisch van het type **Int** of **Double** zoals je gemakkelijkheidshalve zou kunnen veronderstellen. **1** behoort tot de typeklasse **Num** en **1.5** behoort tot de typeklasse **Fractional**. **1** kan met andere woorden zowel een **Int**, **Double**, **Float** of **Integer** zijn. Als niet uit de context duidelijk is welke keuze Haskell moet maken, zul je zelf moeten aangeven tot welk datatype de **1** die je schrijft behoort. Dit is een systeem dat in veel gevallen handig is. Als je in Haskell iets schrijft als **x/2** hoef je niet noodzakelijk **x/2.0** te schrijven. Het nadeel is dat je soms duidelijk moet maken aan de compiler tot welk type het getal dat je schrijft behoort. De compiler geeft de foutmelding **Ambiguous type variable**. Er zijn twee oplossingen. Ofwel schrijf je het type rechtstreeks naast het getal, bijvoorbeeld **1::Int** ofwel geef je het type aan in de signatuur van de functie.

3.3 Lazy evaluation en machtreeksen

Wat volgt is grotendeels gebaseerd op Karczmarczuk [1997]. Een aantal randgevallen zijn hier voor de eenvoud van de voorstelling weggelaten uit de gegeven definities. Die inbrengen is een triviale opdracht.

3.3.1 Machtreeksen

Voorstelling

We zullen machtreeksen als $c_0+c_1x+c_2x^2+\cdots$ voorstellen als lijsten in Haskell. We schrijven **[c0,c1,c2,...]**. In de voorbeelden die volgen kun je meestal ook veeltermen in plaats van machtreeksen gebruiken.

Operaties op machtreeksen

Optellen en aftrekken van machtreeksen is triviaal in Haskell aan de hand van de functie **zipWith**. Die past een gegeven functie toe op de respectieve elementen van twee lijsten en plaatst de resultaten in een nieuwe lijst (zie figuur 3.7a op pagina 54).

Figuur 3.7: Operaties op machtreeksen

(a) Optellen en aftrekken van machtreeksen.

```
u + v = zipWith (+) u v
u − v = zipWith (−) u v

[1,5] + [1,8] = [2,13]
```

(b) Vermenigvuldigen en delen van machtreeksen.

```
(u0:ux) * v@(v0:vx) = (u0*v0):(u0*vx + ux*v)
(u0:ux) / v@(v0:vx) = (w0:wx) where
    w0 = u0/v0
    wx = (ux − w0*vx)/v

[10,4] / [5,2] = [2,0]
```

(c) Integratie en afleiding van machtreeksen.

```
integrate c u = c: zipWith (/) u [1..]
differentiate (u0:ux) = zipWith (*) ux [1..]

integrate 100 [10,2] = [100,10,1]
```

Vermenigvuldigen en delen van machtreeksen is even eenvoudig. Je past bij de deling simpelweg de staartdeling toe. In figuur 3.7b betekent de notatie **v@(v0:vx)** dat **v** een alias is voor de lijst die met **v0:vx** opgevangen wordt.

Als je twee oneindige machtreeksen door elkaar deelt en je wilt slechts de eerste drie termen van het resultaat dan zal lazy evaluation ervoor zorgen dat er niet meer van het algoritme uitgevoerd wordt als nodig.

Ook integratie en afleiding zijn triviale operaties. In figuur 3.7c staat **[1..]** voor **[1,2,3,4,5, . . .]** wat een oneindige lijst is. De **c** in de definitie van **integrate** staat hier voor de integratieconstante.

Toepassing van functies op machtreeksen

Als je $w(x) = f(u(x))$ één keer afleidt en terug integreert krijg je $w(x) = C + \int f'(u(x))u'(x)dx$ en dit geeft de mogelijkheid om recursief operaties op lijsten voor te stellen. Bijvoorbeeld voor de exponentiële functie kun je $exp(u(x)) = \int exp(u(x))u'(x)$ schrijven. In Haskell wordt dit figuur 3.8a op pagina 55. Dit is een recursieve definitie, want het resultaat **w** komt voor in de definitie van het resultaat zelf. Lazy evaluation bevrijdt ons van de kopzorg om rekening te moeten houden met het feit, dat het resultaat impliciet gedefinieerd is aan de hand van het resultaat. Voor elk extra element dat gevraagd is, wordt net zoveel uitgewerkt als strikt nodig.

Van het resultaat weet je dat de constante term de exponentiële is van de oorspronkelijke constante term, vandaar de constante term **exp u0** bij **integrate**. **seriesExp [0,1]** is de expansie van de exponentiële functie.

Op dezelfde manier kunnen zowat alle andere functies toegepast op machtreeksen uitge-

Figuur 3.8: Toepassing van functies op machtreeksen

(a) Exponentiële toegepast op een gegeven machtreeks **u**.

```
seriesExp u@(u0:ux) = w where
  w = integrate (exp u0) (w*differentiate u)

seriesExp [0,1] = [1,1,1/2,1/6,1/24,1/120, ...]
```

(b) Logaritme toegepast op een gegeven machtreeks **u**.

```
seriesLog u@(u0:ux) = w where
  w = integrate (log u0) (differentiate u)/u
```

werkt worden. We geven nog één eenvoudig voorbeeldje, de rest kun je uit een boek over analyse halen. $log(u(x)) = \int \frac{u'(x)}{u(x)}$ geeft figuur 3.8b. Hier is het resultaat zelfs helemaal niet recursief gedefinieerd.

Een wiskundige definitie werkt hier als algoritme. Deze methode zullen we in volgende voorbeelden regelmatig gebruiken. De hoeveelheid werk die voor het resultaat nodig is wordt bepaald door het aantal elementen dat gevraagd is. De rest gebeurt automatisch door lazy evaluation.

Omkering van een machtreeks

Gegeven $z = t+v_2t^2+v_3t^3+\cdots$ wil je t vinden uitgedrukt in z als $t = z+w_2z^2+w_3z^3+\cdots$. Dit vergt enige omwerking voor je het in Haskell kunt schrijven. Je definieert eerst een hulpreeks p zó dat $t = z(1-pz)$. Als je die definitie substitueert in de oorspronkelijke machtreeks z en wat omwerken krijg je een oneindig Hornerschema $p = (1-pz)^2(v_2+z(1-pz)(v_3+z(1-pz)(v_4+\cdots$ en dit is recursief gedefinieerd. Je krijgt code als in figuur 3.9a op pagina 56. Hierbij staat **m** voor $(1 - pz)$ in de definitie voor t. t is gelijk aan $z * m$ en krijg je **t= 0:m**. De rest is vanzelfsprekend. **horner v t** past het Hornerschema toe op de reeks **v** met variabele **t**. Andermaal zorgt lazy evaluation ervoor, dat we ons geen zorgen moeten maken over het feit dat het resultaat gedefinieerd is aan de hand van het resultaat. Door lazy evaluation wordt juist berekend wat nodig is en hoef je niet expliciet rekening te houden met de grenzen van wat je kent en nog moet berekenen bij elk extra element. Haskell doet dit voor ons.

Continue fracties

$$u_0 + u_1x + u_2x^2 + \cdots = g_0 + \frac{g_1x}{1 + \frac{g_2x}{1+\cdots}}$$

wordt figuur 3.9b.

Terugsubstitutie

Met lazy evaluation is terugsubstitutie een eenvoudige zaak. Voor een Padé van een gegeven continue fractie breek je de continue fractie af na $2m$ coëfficiënten. Als je die terug reconvolueert krijg je de $[m/m]$ diagonale Padé benadering. Dit reconvolueren gebeurt met terugsubstitutie wat zeer eenvoudig is in Haskell.

Figuur 3.9: Andere bewerkingen op machtreeksen

(a) Omkering van een machtreeks

```
seriesReverse (0:1:v) = t where
  t = 0 : m
  m = 1:-p
  p = (m^2)* horner v t
```

(b) Continue fracties.

```
continuedFraction (u0:ux) = u0:continuedFraction (1/ux)
```

(c) Padé benadering

```
dpad 0 (g0:_) = ([g0],1)
dpad m (g0:gq) = (g0*p+(0:q),p)
  where (p,q) = dpad (m-1) gq
```

(d) Cayley bomen

```
tau = (0:psi) where
  psi = seriesExp (exsum 1)
  exsum m = 0: ((1/m)*(comPower (m-1) psi) + exsum (m+1))
```

In figuur 3.9c gebeurt het volgende

- Haskell loopt de lijst die als argument meegegeven wordt af tot **m** nul is.
- Vervolgens past Haskell de bewerkingen toe op de gegevens die zo bekomen zijn.

Een eenvoudig voorbeeld

- bij **dpad 1 [1..]** komt in **(p,q) dpad 0 [2..]**
- **dpad 0 [2..] = (2,1)** dus **(p,q) = (2,1)**
- dus **dpad 1 [1..]** is gelijk aan **(1*2+[0,1],2) = ([2,1],2)**

De $[4,4]$ Padé benadering van de exponentiële functie krijg je met **dpad 8 (exp [0,1])**

$$\frac{1+\frac{1}{2}x+\frac{3}{28}x^2+\frac{1}{84}x^3+\frac{1}{1680}x^4}{1-\frac{1}{2}x+\frac{3}{28}x^2-\frac{1}{84}x^3+\frac{1}{1680}x^4}$$

Partitie functies

De generator van Cayley bomen ziet er als volgt uit

$$\tau(x) = x\exp\left(\tau(x)+\frac{\tau(x^2)}{2}+\frac{\tau(x^3)}{3}+\cdots+\frac{\tau(x^m)}{m}+\cdots\right)$$

Er bestaat geen gesloten uitdrukking voor de coefficiënten van τ. Toch zou het nuttig kunnen zijn om een aantal coëfficiënten van τ te kennen. Gezien dit een recursieve definitie is, is dit ideaal voer voor Haskell. In figuur 3.9d staat **psi** voor de hele uitdrukking met de exponentiële

functie. Je past hier de reeds eerder gedefinieerde functie **seriesExp** toe op alles wat tussen de haken staat. Wat tussen de haken staat wordt gegenereerd door de functie **exsum** die telkens een toepassing van τ op een verdere macht van x optelt.

Wiskundig zou je dat kunnen schrijven als $exsum(1) = \tau(x) + exsum(2) = \tau(x) + \frac{\tau(x^2)}{2} + exsum(3) = \cdots$. Merk op dat je zo goed als hetzelfde kunt schrijven in Haskell. Je kunt net zo goed redeneren over wiskundige concepten aan de hand van functies in Haskell als je met de wiskundige schrijfwijze zou doen.

De functie **comPower** substitueert de variabele in een machtreeks met een gegeven macht plus één van die variabele. Dit komt erop neer dat in de lijst een vast aantal nullen tussen elk element van de lijst moet ingevoegd worden. Wiskundig kun je $comPower(1, \tau(x)) = \tau(x^2)$ schrijven.

Oneindig product

Een eindige benadering van

$$Z(x) = \prod_{n=1}^{\infty} \frac{1}{1 - x^n}$$

is lastig. Herschrijf dit als een open recursie door middel van een hulpfunctie $Z_m(x)$

$$Z(x) = Z_1(x), Z_m(x) = \frac{1}{1 - x^m} Z_{m+1}(x)$$

Dit geeft nog geen bruikbaar algoritme. Herschrijf verder als

$$Z_m(x) = Z_{m+1}(x) + x^m Z_m(x)$$

Nu introduceer je een hulpfunctie $B_m(x)$ zó dat $Z_m(x) = 1 + x^m B_m(x)$ en dit geeft

$$B_m(x) = 1 + x(B_{m+1} + x^{m-1} B_m(x))$$

Dit kun je bijna letterlijk ingeven in Haskell

```
partgen = 1 : b 1 where
  b m = 1: b (m+1) + byxn (m-1) (b m)
```

Hierbij is **byxn** een functie die een gegeven machtreeks vermenigvuldigt met een bepaalde macht van een variabele. Gezien machtreeksen als lijsten voorgesteld worden voegt **byxn** een gegeven aantal nullen vooraan de lijst toe.

Conclusie

Lazy evaluation bevrijdt ons van de last om expliciet rekening te moeten houden met de grenzen van potentieel oneindige mathematische objecten bij het vinden van resultaten. Gecombineerd met de functionele mogelijkheden van Haskell laat het toe om direct te werken met de extrapolatie van recursieve problemen in de wiskunde. De keten bij een recursie wordt hierbij direct omgezet in een compacte representatie die direct ook een volledige oplossing is.

Zo ben je meer met wiskunde bezig dan met implementatiedetails. In heel veel gevallen volstaat het om een gegeven wiskundige definitie wat te herwerken tot een recursie of iets wat als lijst handelbaar is. Je kunt het bijna letterlijk overzetten in Haskell. Dat dit allemaal niet nutteloos is blijkt uit de ervaring van Jerzy Karczmarczuk die bij het schrijven van deze

relatief eenvoudige toepassingen een fout ontdekte in een oneindige precisie pakket zonder dat het zijn bedoeling was.

Het is belangrijk om op te merken dat je hier in geen enkel geval met een procedurele specificatie van een algoritme te maken hebt. Telkens gaat het om een wiskundige formulering die bijna letterlijk kan overgezet worden naar Haskell. Aleen moet je soms de wiskundige formulering omvormen tot iets wat je met lazy evaluation te lijf kunt gaan.

3.4 Problemen

Het is niet allemaal goud wat blinkt. Een degelijke kennis van het typesysteem in Haskell is onontbeerlijk voor een goed gebruik van de mogelijkheden die Haskell biedt. We hebben bij de vorige secties bewust een aantal problemen onvermeld gelaten.

Bij type inference hebben we al vermeld dat je niet mag veronderstellen dat **1** tot het datatype **Int** behoort en wat de voor en nadelen hiervan zijn.

Het automatisch omzetten van een datatype naar een ander datatype heeft ook zijn grenzen. De onverwachte problemen betreffen hier vooral variabelen die door het gebruik van een bepaalde functie of operator de beperking hebben dat ze van het type **Int** zijn en door het gebruik van een andere functie of operator tot een klasse moeten behoren waartoe **Int** niet hoort. Het meest voorkomende geval manifesteert zich in het volgende stukje code dat een bepaald aantal elementen van een lijst neemt en die elementen door dat aantal deelt

```
bewerklijst n lijst = fmap (/n) (take n lijst)
```

Door het gebruik van de operator / moet **n** tot de klasse **Fractional** behoren. Door het gebruik van **n** in het eerste argument van **take** moet **n** van het type **Int** zijn. **Int** is evenwel geen instantie van de klasse **Fractional** dus compileert dit niet.

Dit kun je oplossen door één van de functies **fromInteger**, **fromIntegral** en **fromRational** te gebruiken. Het vorige wordt

```
bewerklijst n lijst fmap (/(fromInteger n)) (take n lijst)
```

Iets gelijkaardigs kan ook met **fromRational** en **fromIntegral**. In alle drie de gevallen beantwoordt **Int** aan de beperking die op het argument van de functie rust en zet de functie dat argument om naar iets van het type dat beantwoordt aan de beperking die op het argument van / rust. Handig is anders. Hier werkt het strenge typesysteem van Haskell tegen de programmeur.

Bij de sectie over lazy evaluation kun je niet zomaar de code als in figuur 3.7a op pagina 54 schrijven in Haskell. De operator + hoort tot de typeklasse **Num**. Je moet een instantiedeclaratie maken waarbij lijsten instantie worden gemaakt van de klasse **Num**. Voor de deling moet je lijsten dan weer een instantie maken van de klasse **Fractional**.

Bij de Padé benadering kun je niet exact in Haskell schrijven wat er in figuur 3.9c op pagina 56 staat. Meer bepaald met **g0*p** heeft Haskell problemen. De reden hiervoor is dat **g0** geen lijst is maar **p** wel. De signatuur van de vermenigvuldiging met * is vastgelegd in de typeklasse **Num** wat betekent, dat de vermenigvuldiging alleen werkt op twee variabelen van hetzelfde type. Haskell klaagt over

```
Occurs check: cannot construct the infinite type: a = [a]
  Expected type: a
  Inferred type: [a]
```

```
In the second argument of `(*)', namely `p'
In the first argument of `(+)', namely `g0 * p'
```

Als je C++ gewoon bent, denk je dat je de operator * kunt overloaden om een oplossing te hebben. Overloading werkt in C++ niet op dezelfde manier zoals ze in Haskell werkt. De signatuur van de operator die in **Num** vastgelegd is, kun je niet veranderen. * werkt alleen op twee argumenten van hetzelfde type.

Je hebt in dit geval een aantal opties. Ofwel schrijf je **[g0]** in plaats van **g0**, ofwel definieer je een operator specifiek om een gewoon getal te vermenigvuldigen met een lijst, ofwel laat je Haskell *automatisch* **g0** omzetten in een lijst. Dit laatste gebeurt door **fromRational g0** te schrijven in plaats van **g0**. Geen van de drie oplossingen zijn handig en de laatste oplossing is zeker niet altijd een bruikbare oplossing. **fromRational** moet in dit geval gedefinieerd zijn op lijsten zodat een variabele omgezet wordt door **fromRational** in een lijst met alleen die variabele in. Wat de verwarring voor de Haskell beginner nog groter maakt is, dat Haskell de laatste oplossing automatisch doorvoert als je **4 * [1,2,3]** aan de interactieve commandline probeert. Dit automatisme is ingevoerd om gebruikers van de interactieve commandline niet te verplichten telkens het type te moeten specifiëren in gevallen waar een oplossing voor de hand ligt.

Een gelijkaardig probleem duikt op bij de Cayley bomen. Als je voor de code in figuur 3.9d op pagina 56 **comPower** implementeert zodat het eerste argument van het type **Int** moet zijn, kom je in de problemen. De deling is slechts gedefinieerd voor getallen van de klasse **Fractional** en de compiler klaagt over:

```
No instance for (Floating Int)
  arising from use of `serExp' at lijstversie.hs:152:8-13
Probable fix: add an instance declaration for (Floating Int)
In the definition of `psi': psi = serExp (exsum 1)
In the definition of `tau':
    tau = (0 : psi)
        where
            psi = serExp (exsum 1)
            exsum m = 0:(([1/m] * (comPower (m-1) psi))+(exsum (m+1)))
```

Wat is hier aan de hand? **m** is door het gebruik van **comPower** van het type **Int**. Je hebt hier met hetzelfde probleem te maken als we bij type inference vermeld hebben. Dit zien is voor een beginnende Haskell programmeur zeker niet evident, omdat je je door het automatisch laten bepalen van de signatuur van **comPower** niet realiseert wat het probleem is.

De volledig werkende versie van de hand van Karczmarczuk kunt u vinden op `http://www.reinaert.albrecht.easynet.be/ithesis/`. Ik kreeg die, nadat ik hem attent had gemaakt op een aantal problemen die ik had bij de implementatie van wat in zijn paper stond. Op dat moment had ik het grootste stuk al werkend geïmplementeerd. Mijn problemen hadden in hoofdzaak te maken met mijn onkunde op het vlak van het typesysteem in Haskell op dat moment. Dit werd nog gecompliceerd door het gebruik van de Haskell compiler Hugs. De typechecker van Hugs is minder goed dan die van the Glasgow Haskell Compiler (GHC, zie GHC [2005]). De foutmeldingen zijn minder duidelijk en GHC lost meer type-problemen op die Hugs niet kan aanpakken. We hebben vanaf dat moment alleen nog GHC gebruikt voor ontwikkeling.

Patternmatching niet meer handig als je een datatype met veel velden hebt. Als de velden niet allemaal door elke functie gebruikt worden of de kans is groot dat tijdens de ontwikkeling nieuwe velden aan het type toegevoegd zullen worden, dan kan je beter gebruik maken van veldlabels (zie sectie 3.1.6 op pagina 44). Je vermijdt op die manier fouten in patternmatching tegen de positie in een datatype. Je vermijdt ook dat, bij het toevoegen van een nieuw veld, je alle patternmatching op het datatype moet aanpassen in het volledige programma. Patternmatching gebruik je best op datatypes met weinig velden die weinig kans hebben aangepast te worden tijdens de ontwikkeling. Veldlabels gebruik je best op heel uitgebreide datatypes, zeker als die extra velden kunnen krijgen.

3.5 Conclusie

Waarvoor zijn functionele talen als Haskell in de context van numerieke software nuttig? Ze besparen ons heel wat werk op een aantal vlakken. Het concept van functies in Haskell ligt veel dichter bij het concept van functies in de wiskunde dan in niet functionele talen. Je kunt ook eenvoudig in de taal over functies en hun types redeneren. Het concept van typeklassen en hoe je daarmee modellen opbouwt ligt dicht bij hoe je in de wiskunde omgaat met verzamelingen en groepen. Het gebruik van pattern matching ligt dichter bij hoe je over structurering van algoritmes nadenkt. Zo vermijd je vervelende *if-then-else*-structuren en omslachtige constructie en deconstructie van structuren. Tegelijkertijd is sneller duidelijk wat geschreven staat en het algoritme wordt logisch opgesplitst. Als je met potentieel oneindige structuren moet omgaan is lazy evaluation zo goed als onmisbaar. Hetzelfde in een taal doen die dit niet ondersteunt is uitermate lastig qua implementatie, waardoor je meer tijd besteedt aan implementatiedetails dan aan het effectief oplossen van het probleem.

Ondanks al deze voordelen zijn er ook nadelen. In de eerste plaats is de snelheid van Haskell-code een stuk trager dan code die hetzelfde resultaat heeft in een imperatieve taal. Voor zware *number crunching* zal je Haskell in elk geval niet gebruiken. Waar de modellering van het probleem in een imperatieve taal moeilijk en omslachtig wordt, verdwijnt dit nadeel. Bijvoorbeeld voor de simulatie van een quantumcomputer (zie Karczmarczuk [2003], Sabry [2003] en Skibinski [2001]). Voor het programmeren van een dergelijke, voorlopig nog hypothetische, computer is er onderzoek naar het nut van functionele talen. Zo'n computer werkt op een heel andere manier dan een klassieke computer en *klassieke* talen blijken niet zo geschikt om zo'n computer te programmeren. Zie bijvoorbeeld Selinger [2004], van Tonder [2003], Selinger and Valiron [2004], Selinger [2003] en Grattage and Altenkirch [2005].

Een ander nadeel is dat het mengen van pure functies en niet-pure functies, bijvoorbeeld voor IO, niet altijd vanzelfsprekend is. Soms verlies je veel van de gewonnen tijd voor iets wat je in een imperatieve taal als een vanzelfsprekendheid beschouwt.

Het is spijtig dat studenten wiskunde, die constant met functies bezig zijn, in hun curriculum doorgaans nooit in contact komen met een functionele taal. Men zou kunnen argumenteren dat een taal als Maple functioneel is. Dit klopt, maar de functionele mogelijkheden van Maple zijn zeer beperkt. Mathematica is wel een functionele taal, maar wordt tijdens de opleiding zelden of nooit gebruikt.

Samenvattend ligt de manier waarop je een wiskundige uitdrukking implementeert in een functionele taal conceptueel dichter bij de oorspronkelijke uitdrukking dan bij niet functionele talen. Daar staat tegenover dat functionele talen tragere programma's opleveren dan een vergelijkbaar programma in een imperatieve taal.

Hoofdstuk 4

Vergelijking C++ en Haskell

4.1 Implementatie van problemen

Om de vergelijking te staven implementeerden we twee problemen in de talen C++ en Haskell.

4.1.1 Diffusieprobleem

Omschrijving probleem

We kozen voor de oplossing van de tweedimensionale warmtevergelijking

$$\frac{\partial u}{\partial t} = \kappa(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) \tag{4.1}$$

door evolutie in de tijd. Hierbij geeft $u(t, x, y)$ de temperatuur in functie van tijd en positie in het vlak. κ is de warmtegeleidingscoëfficiënt van het materiaal. Deze vergelijking kan numeriek opgelost worden door discretisatie in tijd en ruimte. Je vervangt hierbij elke partiële afgeleide door een eindige differentie. We gebruiken een tweede orde centraal differentieschema waarbij

$$\frac{\partial^2 u_{i,j}}{\partial x^2} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x} + O(\Delta x)^2. \tag{4.2}$$

Hierbij is Δx de discrete stap in de ruimte. Vergelijking 4.1 wordt

$$u_{x,y}^{t+1} = u_{x,y}^{t} + D\left(u_{x-1,y}^{t} - 2u_{x,y}^{t} + u_{x+1,y}^{t}\right) + E\left(u_{x,y-1}^{t} - 2u_{x,y}^{t} + u_{x,y+1}^{t}\right) \tag{4.3}$$

waarbij

$$D = \frac{\kappa \Delta t}{\Delta x^2}$$
$$E = \frac{\kappa \Delta t}{\Delta y^2}.$$

Als Δt aan de Courant-Friedrichs-Levy-conditie voldoet, dan is deze oplossingsmethode stabiel. Dit betekent dat Δt aan de ongelijkheid

$$\Delta t < \frac{(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2})^{-1}}{\kappa}$$

moet voldoen.

Analyse

Als je met een rechthoekig domein te maken hebt, kun je de discrete puntenwaarden opslaan in een matrix. We willen ook de mogelijkheid om onregelmatige domeinen op te geven. We behouden het gebruik van een matrix en voeren een functie in om te bepalen welke punten van die matrix tot het domein horen en welke niet. We noemen deze functie de domeinfunctie. De punten uit de matrix die niet tot het domein behoren zijn dan randpunten. De randpunten van de matrix mogen niet tot het domein behoren omwille van het iteratieschema. We zullen in de code vastleggen dat de randpunten niet tot het domein behoren ongeacht wat als domeinfunctie opgegeven is.

Het domein moet, voor het itereren begint, gevuld worden met startwaarden. De randvoorwaarden houden we statisch en moeten slechts één keer in de matrix gezet worden, omdat het iteratieschema geen enkele randvoorwaarde aanpast. Dit doen we met behulp van twee geleverde functies.

We willen drie methodes hebben om iteratiestappen te zetten:

- Eén enkele iteratiestap zetten.
- Itereren gedurende een bepaalde tijd wat gelijk is aan het zetten van $\frac{tijd}{\Delta t}$ iteratiestappen.
- Itereren tot een bepaalde voorwaarde vervuld is.

Voor de eenvoud zullen we voor de laatste methode als voorwaarde een maximale verandering in het domein gebruiken. Zolang er bij één punt uit het domein een verandering is, die groter of gelijk is aan de maximale verandering die opgegeven is, blijven we itereren. Δx stellen we gelijk aan Δy. We zullen in het differentieschema waarden uit de huidige iteratie gebruiken. We gebruiken in plaats van het differentieschema in 4.3 volgend schema

$$u_{x,y}^{t+1} = u_{x,y}^{t} + D\left(u_{x-1,y}^{t+1} + u_{x+1,y}^{t} + u_{x,y-1}^{t+1} + u_{x,y+1}^{t} - 4u_{x,y}^{t}\right). \tag{4.4}$$

Samenvattend zijn de parameters van het probleem:

- κ, Δx en Δt uit de vergelijking 4.4
- de coördinaten van de linkerbenedenhoek en rechterbovenhoek
- een domeinfunctie
- een vulfunctie voor de startwaarden in het domein en een functie voor de randvoorwaarden.

Tot slot hebben we een mogelijkheid nodig om de waarden uit de matrix in een bestand te plaatsen zodat we de gegevens kunnen visualiseren.

Objectgericht ontwerp

Een klassendiagram vindt u in figuur A.1 op pagina 85. Het diffusieprobleem bevatten we volledig in één klasse. Bij de initialisatie worden alle parameters opgegeven. We hebben een matrix (**_werkmatrix**) als attribuut nodig om alle waarden in op te slaan. We hebben drie methodes nodig om iteratiestappen te zetten (**zetIteratiestap**,**itereerTijd**, **itereerTotEinde**). We hebben een methode nodig om de waarden uit de matrix in een bestand te plaatsen (**toon**). We hebben een methode nodig om de tijd te geven die verstreken is sinds het begin van de simulatie (**geefTijd**). We hebben dus ook een attribuut nodig dat het aantal stappen dat gezet is bijhoudt (**_aantalIteraties**). Intern hebben we twee functies nodig, die een x-

en y-positie in de matrix omzetten naar de echte waarden in het domein (**naarxPositie** en **naaryPositie**). Ook hebben we een methode nodig, die controleert of de opgegeven positie tot het domein behoort, en ervoor zorgt dat de randen van de matrix niet tot het domein behoren, ongeacht welke domeinfunctie de gebruiker opgeeft (**hoortTotDomein**).

C++

Een overzicht van de methodes vindt u in tabel A.1 op pagina 86. Voor het opslaan van de waarden in een matrix gebruiken we de Blitz++ bibliotheek (Veldhuizen et al. [2005]). Voor het doorgeven van functies gebruiken we het template mechanisme in C++ omwille van de redenen die je in sectie 2.2.1 op pagina 20 vindt. We initialiseren het domein en de randvoorwaarden bij de objectcreatie.

De methode **toon** plaatst de waarden uit de volledige array in een bestand zodat die, onder andere met GnuPlot, in een grafiek omgezet kan worden. We houden in een klassevariabele bij wat de maximale verandering in het domein is na elke iteratie.

De code van het programma vindt u op pagina 84. Een typische use case van het programma vindt u vanaf regel 315 op pagina 91.

Functioneel ontwerp

Encapsulatie bestaat niet in Haskell. We zullen alle parameters en gegevens die we bij moeten houden in een datatype moeten bevatten en dit gegeven telkens door moeten geven aan de functies die iets met het probleem doen. Voor de rest gelden dezelfde ontwerpbeslissingen als bij het objectgericht ontwerp. Alleen worden alle methodes die het object veranderen, functies die een datatype veranderen.

Haskell

Een overzicht van de functies vindt u in tabel A.2 op pagina 92. Er is geen wezenlijk verschil tussen de equivalente methodes in de C++ versie, behalve dat deze functies als extra parameter het datatype met de probleembeschrijving krijgen. Het datatype (zie regel 27 op pagina 93) maakt gebruik van veldlabels, omdat er veel gegevens in opgeslagen worden en het datatype eenvoudig aan te passen moet zijn, zonder dat je alle pattern matching erop moet aanpassen.

De code van het programma vindt u op pagina 91. Een typische use case van het programma vindt u vanaf regel 159 op pagina 95. Merk op dat het programma alleen werkt met GHC, niet met Hugs.

Resultaat

In figuur 4.1 op pagina 66 en figuur 4.2 op pagina 67 vind je de resultaten van beide programma's voor twee verschillende cirkelvormige domeinen bestaande uit koper en aluminiumoxyde. Koper heeft ongeveer $1.1161\frac{cm^2}{s}$ als waarde voor κ. Aluminiumoxyde heeft ongeveer $0.0879\frac{cm^2}{s}$ als waarde voor κ (Hukseflux [2005]). Het proces convergeert duidelijk sneller voor een koperen schijf dan voor een schijf uit aluminiumoxyde.

Korte vergelijking

Voordelen van de C++ versie ten opzichte van de Haskell versie:

- Veel sneller. Voor de eerste simulatie loopt het programma op een Pentium 4, 2 GHz gecompileerd met g++ versie 4.0.3 en optimalisatie **-O2** in 0.15 seconden. De Haskell versie doet over dezelfde simulatie met GHC 6.4.1 en optimalisatie **-O2** 48.26 seconden.
- De notatie voor indices van een matrix is natuurlijker: vergelijk **t(i,j)** met **t!(i,j)**.
- Matrix notatie is duidelijker dan in Haskell. Vergelijk **Array<double,2>** met **a (i,i) e**. In principe hoeven we de Haskell notatie nergens te schrijven, maar je zal ze wel tegenkomen bij foutmeldingen.
- Optimalisatie is veel eenvoudiger in C++. In Haskell is het type array, het feit of je waarden uit de matrix die je opbouwt gebruikt of niet, en de manier waarop je itereert van invloed op hoe snel de code gaat en hoeveel geheugen ze gebruikt.
- Alle code voor een warmteprobleem zit in een klasse. In Haskell moet je met modules werken en binnen die modules is er altijd een scheiding tussen de interface en de data waarop die interface werkt. In Haskell geef je het probleem door. Dit betekent dat je in Haskell telkens de gegevens uit het argument van een functie moet halen. In C++ zitten die in het object en je kunt er direct naar verwijzen.
- De afhankelijkheden tussen de opeenvolgende gegenereerde matrices bepaalt in Haskell hoeveel geheugen gebruikt wordt. In onze simulatie gebruikt het programma meer dan 300 megabyte geheugen. Dit is in C++ helemaal niet het geval.
- Het berekenen van de grootste verandering per iteratiestap kan handiger in C++ (zie regel 196 op pagina 89) dan in Haskell (zie regel 137 op pagina 95). In C++ volsta je met twee toekenningen, doordat je eenvoudig losse statements tussen code kunt schrijven. In Haskell is de eenvoudigste oplossing om de volledige matrix in één keer te overlopen.

Voordelen van de Haskell versie ten opzichte van de C++ versie:

- In C++ ervoor zorgen dat je eenvoudig het updateschema kunt aanpassen zorgt ervoor, dat de code voor het zetten van een iteratiestap moeilijker leesbaar en langer wordt (zie regel 168 op pagina 88). Je moet expliciet de vorige matrix kopiëren en dan aan de hand daarvan de huidige opbouwen, wat redelijk wat aanpassingen in de code vergt. In Haskell (zie regel 168 op pagina 88) gebruik je ofwel alleen de matrix die binnenkomt, ofwel gebruik je voor bepaalde waarden de matrix die je opbouwt in de uitdrukking waar je ze opbouwt.
- Doorgeven van functies is eenvoudig.
- Als je over de grenzen van de matrix gaat krijg je een foutmelding tijdens de uitvoering. In C++ helemaal niet, tenzij het programma vastloopt, wat niet altijd gebeurt.
- Geen gebruik van pointers dus minder kans op programmeerfouten.
- Geen automatische conversie van het ene ingebouwde type naar het andere. In C++ kun je op regel 155 op pagina 88 de functie **round** weglaten en de code zal foutief compileren. In Haskell moet je expliciet omzetten van het ene type naar het andere, anders compileert je programma niet.
- Voor het schrijven van eenvoudige functies om domein en randvoorwaarden op te geven kun je lambdafuncties gebruiken. In C++ moet je een nieuwe functie schrijven.
- De interface is voor alle soorten arrays gelijk in Haskell. Het vraagt geen aanpassingen aan de code om van arraytype te veranderen.

- Door de lazyness van Haskell kun je uitdrukkingen die mekaar definiëren schrijven zonder dat dit een probleem is. De definitie van **probleem** (zie regel 185 op pagina 96) gebruikt de definitie van **start** (zie regel 178 op pagina 95) en omgekeerd.

Het schrijven naar bestanden is ongeveer evenveel werk in beide talen.

4.1.2 Lattice Rules

Omschrijving probleem

Lattice rules zijn een quasi Monte Carlo methode. De rule of regel is de integratieregel. De lattice bepaalt welke discrete puntenset gebruikt wordt.

Een vectoriële lattice is de volgende verzameling vectoren

$$\left\{ \frac{k * \mathbf{z}}{n} | k \in \mathbb{Z}_n \right\}, \qquad \mathbf{z} \in \mathbb{Z}_n^s. \tag{4.5}$$

Hierbij noemen we $\mathbf{z}$ de generator. Die is vast voor elke lattice. Het symbool $*$ is de scalaire vermenigvuldiging op $\mathbb{Z}_n^s$.

Een polynomial lattice is de volgende verzameling vectoren

$$\left\{ \frac{k * \mathbf{z}}{f(x)} | k \in \mathbb{Z}_b[x]/f(x) \right\}, \qquad \mathbf{z} \in \left(\mathbb{Z}_b[x]/f(x)\right)^s. \tag{4.6}$$

Hierbij is $f(x)$ een irreducibele veelterm in $\mathbb{Z}_b[x]$ en $*$ de scalaire vermenigvuldiging van een polynoom met een vector van polynomen in $(\mathbb{Z}_b[x]/f(x))^s$. We schrijven een quotiëntring $\mathcal{R}/(n\mathcal{R})$ met $n \in \mathcal{R}$ verkort als $\mathcal{R}/n$.

De deling die gebruikt wordt is niet de deling in de verzameling waarover de vectorruimte gedefinieerd is. In het 1-dimensionale vectoriële geval over het veld $\mathbb{Z}_n$ geeft de deling een resultaat in $[0..1[\subset \mathbb{Q}$. Het vectoriële geval over de ruimte $\mathbb{Z}_n$ geeft een resultaat in $[0..1[^s \subset \mathbb{Q}^s$. Het polynomiale geval over het veld $\mathbb{Z}_b[x]/f(x)$ geeft een resultaat in $\mathbb{Z}_b[[x]]$. De ruimte $\mathbb{Z}_b[[x]]$ is de ruimte van de formele Laurent series over $\mathbb{Z}_b$. De deling in het polynomiale geval is een expansie in negatieve machten van de deling van een veelterm van een kleinere graad dan $f(x)$ met de veelterm $f(x)$. Bij het polynomiale geval is nog een extra functie nodig. Die mapt het resultaat na de deling terug naar de vectorruimte die we nodig hebben om te integreren. Dit zal naar $[0..1[^s \subset \mathbb{Q}^s$ zijn.

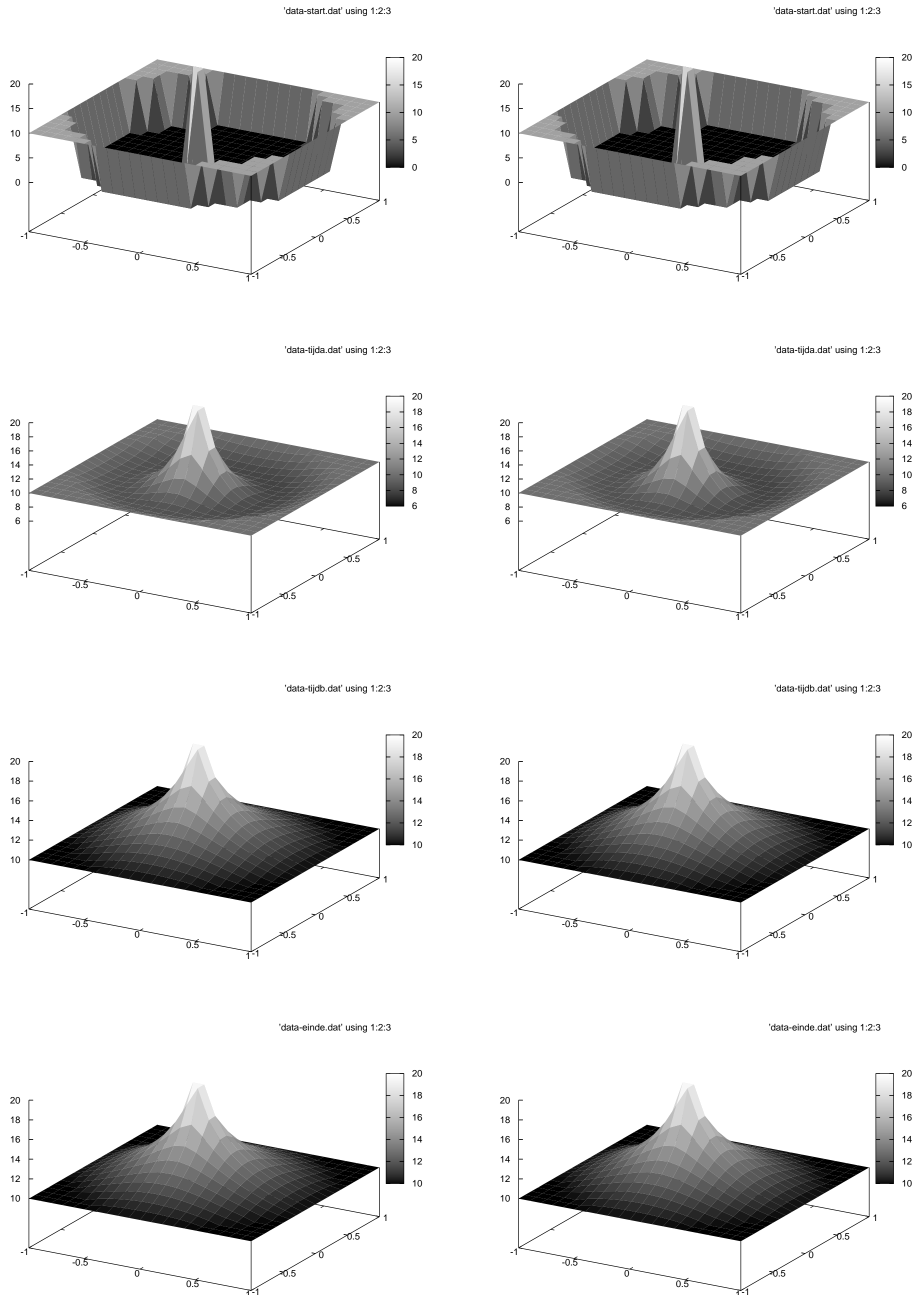
Analyse

Al de lattices veralgemenen tot volgende definitie van een lattice

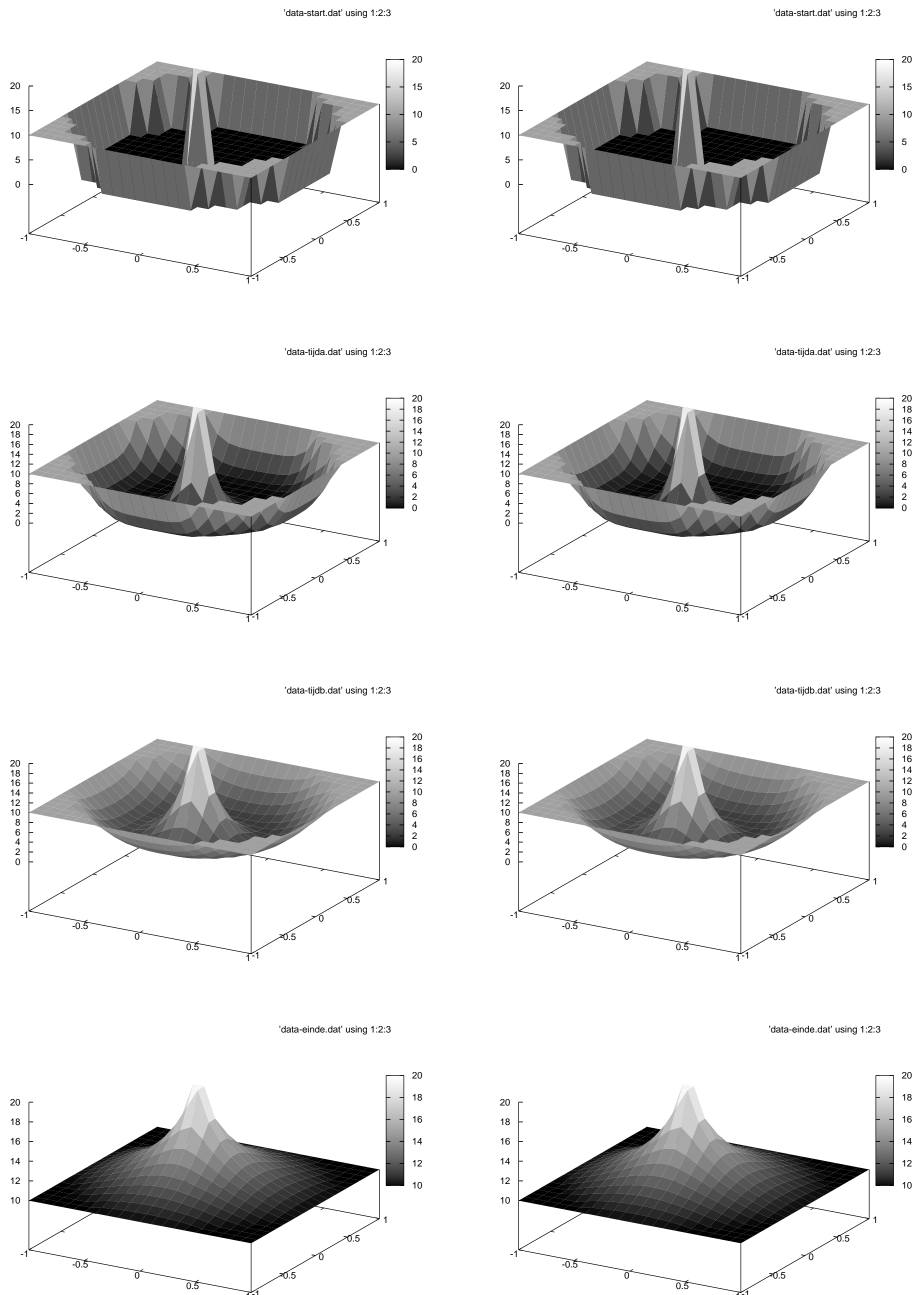
$$\left\{ \frac{k * \mathbf{z}}{n} | k \in \mathcal{E}/n \right\}, \qquad \mathbf{z} \in (\mathcal{E}/n)^s. \tag{4.7}$$

Hierbij is

- $\mathcal{E}$ is een euclidisch domein.
- $(\mathcal{E}/n)^s$ een vectorruimte met dimensie s over de quotiëntring $\mathcal{E}/n$. In het scalaire geval is $s = 1$.
- $\mathbf{z}$ is de generator en behoort tot $(\mathcal{E}/n)^s$.



Figuur 4.1: Oplossing van de warmtevergelijking voor een cirkel in koper met starttemperatuur 0 graden, temperatuur rond de cirkel van 10 graden en een warmtepunt in het midden van 20 graden. Vertikaal is de temperatuur aangegeven, horizontaal de positie in het vlak.Links staan de resultaten van het C++ programma en rechts van het Haskell programma. Van boven naar beneden zijn respectievelijk de startgegevens, de resultaten na 100 milliseconden, de resultaten na 400 milliseconden en de resultaten bij convergentie weergegeven. De convergentievoorwaarde is hier een minimale verandering van 0.0001 graad per iteratiestap. De schijf heeft een diameter van 2 cm. De tijdsstap bedroeg 0.001 seconde. Convergentie werd in beide gevallen bereikt na 0.6 seconden.



Figuur 4.2: Oplossing van de warmtevergelijking voor een cirkel in aluminiumoxyde met starttemperatuur 0 graden, temperatuur rond de cirkel van 10 graden en een warmtepunt in het midden van 20 graden. Vertikaal is de temperatuur aangegeven, horizontaal de positie in het vlak. Links staan de resultaten van het C++ programma en rechts van het Haskell programma. Van boven naar beneden zijn respectievelijk de startgegevens, de resultaten na 100 milliseconden, de resultaten na 400 milliseconden en de resultaten bij convergentie weergegeven. De convergentievoorwaarde is hier een minimale verandering van 0.0001 graad per iteratiestap. De schijf heeft een diameter van 2 cm. De tijdsstap bedroeg 0.001 seconde. Convergentie werd bereikt na 6.109 seconden.

- $*$ is de scalaire vermenigvuldiging in de vectorruimte $(\mathcal{E}/n)^s$ en $k * \mathbf{z}$ kan gelezen worden als $k * \mathbf{z}$ modulo n.
- k loopt in de implementatie alle elementen van $\mathcal{E}/n$ af.

Het resultaat van de deling behoort tot een andere verzameling afhankelijk van de verzameling $\mathcal{E}$. De verzameling waartoe de teller behoort, is altijd een vectorruimte over een quotiëntring. De verzameling waartoe de noemer behoort is altijd het euclidisch domein waarover de quotiëntring van de teller gedefinieerd is.

We maken gebruik van het feit dat de opvolgers van k normaal terug op het nulelement uitkomen om de latticegeneratie te beëindigen. De deling op $\mathbb{Z}_b[x]$ beperken we voor de eenvoud tot een deling door een monische polynoom, omdat we die deling alleen gebruiken bij de modulo operatie op $\mathbb{Z}_b[x]/f(x)$ en $f(x)$ is altijd irreducibel.

De polynomen de we gebruiken in het polynomiale geval hebben alleen positieve machten. Het resultaat van de deling geeft een Laurent serie die alleen negatieve machten heeft. Het enige onderscheid tussen beide, los van het feit dat een Laurent serie negatieve machten kan hebben, is dat een polynoom een eindig aantal coëfficiënten heeft en een serie een oneindig aantal coëfficiënten kan hebben. Een andere mogelijkheid is om het algemene geval van een Laurent serie op te splitsen in een datatype dat uit drie delen bestaat:

- het deel met de strikt negatieve machten
- de coëfficiënt van de nulde macht
- het deel met de strikt positieve machten

Dit is te algemeen voor wat we nodig hebben en zou het schrijven van code voor dit probleem omslachtiger maken.

Functioneel ontwerp

Omdat bij dit probleem veel ideeën uit de groepentheorie gebruikt worden, ligt het voor de hand de klassendefinities van Jeroen Fokker uit sectie 3.2 op pagina 48 te gebruiken. Voor elementen van de quotiëntring $\mathcal{E}/n$ kunnen we wel een typeklasse definiëren, maar dan zouden we slechts één n per verschillende ruimte $\mathcal{E}$ kunnen hebben. Dit is niet wat we willen. Voor een element van een quotiëntring zullen we een datatype invoeren dat het element uit $\mathcal{E}$ bevat en n. We zullen de term modulonummer hanteren voor het element uit $\mathcal{E}$ en de term modulogetal voor n.

Aan de klassendefinities zijn een paar wijzigingen nodig als we met een dergelijk datatype willen werken. In de klasse **Monoid** en **Ring** kunnen we **zero** en **one** niet geven omdat n niet vastgelegd is in **Monoid** en **Ring** en op geen enkele manier uit de context kan achterhaald worden. We vervangen deze functies door de functies met één argument **iszero** en **isone**. Deze controleren of het argument een nul-element is of een eenheid is.

Polynomen en gehele getallen maken we elementen van een euclidisch domein. Elementen van een quotiëntring moeten tot een ring behoren. De ruimte van het resultaat van deling leggen we vast met een typeklasse met twee typeparameters. Om alle elementen k te genereren van een lattice zullen we een opvolgersfunctie nodig hebben. Als deze opvolgersfunctie terug op het nulelement uitkomt zal de generatie van de lattice ten einde zijn. We hebben opvolgers nodig van gehele getallen, elementen van een quotiëntring en van polynomen.

Haskell

Een overzicht van alle functionele componenten van het programma vindt u in tabel A.2.2 op pagina 110. We voeren een **Modulo** datatype in met als tweede veld n (zie regel 203 op pagina 114). Een element van een quotiëntring schrijf je als **Mod element n**. Het nulelement van $\mathbb{Z}_2$ schrijf je bijvoorbeeld als **Mod 0 2**.

We gebruiken een recursief datatype voor de voorstelling van polynomen. Een voorstelling waarbij we alleen de niet-nul coëfficiënten bewaren samen met de macht laat niet toe om op een lazy manier de deling resulterende in een Laurent serie uit te voeren. Polynomen zullen we voorstellen als een serie zonder teken (zie regel 82 op pagina 112), het resultaat van de deling bij het polynomiale geval als een serie met een negatief teken (**Serie Negatief Polynoom**).

Om opvolgers van k te genereren, gebruiken we de functie **succ** uit de Haskell klasse **Enum**. Het is niet noodzakelijk voor elk type een volledige instantie declaratie van **Enum** te geven. De definitie van **succ** volstaat. In het polynomiale geval zal de opvolgersfunctie niet uitkomen op het nulelement en zal de lattice niet eindig zijn. Stel dat we in de quotiëntring $\mathbb{Z}_2[x]/(1+x+x^3)$ werken en de opvolger van $1+x+x^2$ vragen. De definitie van regel 208 op pagina 114 wordt aangeroepen. Hierbij wordt een opvolger van het element $1+x+x^2 \in \mathbb{Z}_2[x]$ gevraagd. Daarna wordt de modulo-operatie toegepast om het element te reduceren. De opvolger van $1+x+x^2$ is x^3. De modulo-operatie hierop toepassen geeft $1+x$. We zullen bijgevolg nooit op het nulelement uitkomen. Haskell laat niet toe een overlappende instantiedeclaratie te geven (bijvoorbeeld voor **Enum (Modulo (Polynoom a))**) zodat dit correct werkt.

Alle instantiedeclaraties voor gehele getallen, polynomen en elementen van een quotiëntring vindt u met hun bijbehorende hulpfuncties vanaf regel 47 op pagina 111 tot regel 231 op pagina 114.

Vectoren kunnen we schrijven als lijsten in Haskell. Over lijsten voeren we de scalaire vermenigvuldiging in met de nieuwe operator **.*** (zie regel 236 op pagina 114). Voor de deling voeren we een nieuwe typeklasse **DivisionType** in met twee parameters (zie regel 242 op pagina 115). De eerste parameter is de ruimte van de noemer en de tweede parameter is de ruimte van het resultaat van de deling. Deze typeklasse voert de delingsoperator `</>` in. Om de verschillende soorten delingen per ruimte vast te leggen, dient een instantie van **DivisionType** gegeven te worden. Merk op dat we overloading op returntype gebruiken voor het resultaat van `</>`. Het type van het resultaat zal steeds opgegeven moeten worden.

De code van het programma vindt u op pagina 109. Het programma werkt vanaf de interactieve commandolijn van zowel Hugs als GHC. Een typische usecase vindt u vanaf regel 324 op pagina 116.

Objectgericht ontwerp

Een klassendiagram vindt u in figuur A.2 op pagina 97. Een modellering van algebraïsche groepen is onzinnig zonder typeklassen. Alle concepten aangaande een lattice groeperen we in een klasse **Lattice**. Het type van de euclidische ruimte waarover gewerkt wordt is geparametriseerd met **ModuloType**. Het resultaat van de deling is geparametriseerd met **ResultaatType**. Elementen van een quotiëntring en hun operaties plaatsen we in een klasse **Modulo**. Polynomen en formele Laurent series plaatsen we respectievelijk in de klasse **Polynoom** en **Serie**. Deze drie klassen zijn geparametriseerd met het type waarover ze moeten gedefinieerd worden. Een polynoom over elementen van een quotiëntring zal bijvoorbeeld als

parameter **Type** de klasse **Modulo** meekrijgen.

C++

Een overzicht van de methoden vindt u in tabel A.3 op pagina 98. Vectoren schrijven we met de STL-container **vector**. Voor de vectoriële vermenigvuldiging overloaden we de *-operator. De interne representatie van polynomen gebeurt met de STL-container **vector**. Wat de opvolgers van polynomen betreft geldt dezelfde opmerking als bij de Haskell versie.

De code van het programma vindt u op pagina 96.

Korte vergelijking

Voordelen van de C++ versie ten opzichte van de Haskell versie:

- Je kunt stoppen in het midden van de generatie van een lattice en eventueel later verder gaan.
- Je kunt bij de creatie van een element van een quotiëntring afdwingen dat het modulonummer *kleiner* is dan het modulogetal.

Voordelen van de Haskell versie ten opzichte van de C++ versie:

- Show functies voor nieuwe types zijn veel eenvoudiger dan de vergelijkbare methode in C++ met overloading van de <<-operator. Vergelijk bijvoorbeeld de code vanaf regel 525 op pagina 107 met de code vanaf regel 134 op pagina 113. Daarnaast is er geen gevaar om per ongeluk fouten met pointers te krijgen (zie sectie 4.3.5 op pagina 76). Eventueel kunnen we Haskell het tonen van een nieuw datatype zelf laten afleiden (zie regel 203 op pagina 114).
- Notatie voor types is veel kernachtiger in Haskell met typeklassen en instanties dan met het gebruik van templates in C++, zelfs al modelleren we in Haskell veel meer dan in C++.
- Geen onverstaanbare compileerfouten zoals je in C++ krijgt met het gebruik van templates.
- Het typesysteem van Haskell is veel eleganter door het deel maken van een datatype aan een typeklasse waar je in C++ je datatype moet boxen en unboxen met een nieuwe klasse. Doordat het datatype zo eenvoudig is, werk je in Haskell met patternmatching wat de code duidelijker en bondiger maakt.
- Patternmatching op recursieve types leest vlotter en geeft minder kans op fouten in vergelijking met het gebruik van containeroperaties in C++ (zie sectie 4.4.1 op pagina 77).
- Het type van het resultaat van een functie wordt door Haskell zelf afgeleid. Je moet resultaattypes niet herhalen zoals bij de methodes in C++, waar je alle types met template parameters erbij voor elke methode moet schrijven.
- Geen nood om de code te hercompileren telkens je functies wilt uitproberen met nieuwe waarden.
- Fouten tegen types krijg je direct bij compilatie van abstracte code. In C++ krijg je de meeste fouten pas bij het effectief gebruik van template code .
- Het is eenvoudiger om vectoren te schrijven in Haskell dan vectorinitialisatie in C++ te doen.

- Bij de deling die een Laurent serie als resultaat heeft kunnen we lazy evaluation (zie regel 255 op pagina 115) gebruiken en hoeven we geen grens op te leggen zoals we dat in C++ doen (zie regel 561 op pagina 108 en regel 480 op pagina 106).

4.2 Functies

4.2.1 Typespecificatie

Het functieconcept in C++ en Haskell is grondig verschillend. Zo goed als alle functies in Haskell kun je zonder typesignatuur schrijven. Dit is vooral handig als het over eenvoudige functies gaat. Je kunt snel korte functies schrijven, zonder dat je tijd verliest met het specifiëren van het type van de functie. Functies zonder typesignatuur zijn direct zo algemeen mogelijk. Zolang in de functie alle gebruikte operatoren en functies op de argumenten kunnen toegepast worden, zal de functie bruikbaar zijn. In C++ kun je niet anders dan altijd de types op te geven, of je nu het type parametriseert of niet. Parameteriseren vraagt extra werk, daar waar veralgemenen in Haskell juist minder werk vraagt.

4.2.2 Lambdafuncties en de interactieve commandolijn

Lambdafuncties kun je gebruiken waar je eenvoudige functies eenmalig nodig hebt. In het diffusieprogramma moet je in C++ de verschillende functies telkens eerst afzonderlijk definiëren (zie regel 286 op pagina 90) voor je ze kunt gebruiken als argument. In de Haskell versie schrijven we die functies ter plaatse waar we ze nodig hebben (zie regel 178 op pagina 95). Je kunt zo ook snel functies uittesten van een reeds gecompileerd programma aan de interactieve commandolijn. Op die manier kun je snel je programma testen, zonder dat je telkens opnieuw de code moet aanpassen, compileren en uitvoeren, zoals je dat in C++ moet doen voor functies.

4.2.3 Structuur

Haskell code bevat hoofdzakelijk geneste uitdrukkingen. Hierdoor wordt de logische structuur van een functie duidelijker. Vergelijk de code in figuur 3.1 op pagina 43 met de code in figuur 2.10b op pagina 38. In de Haskell versie is direct duidelijk welke matrices afhankelijk zijn van waarden uit de vorige iteratie en welke van waarden uit de huidige iteratie. De Haskell versie is net iets langer maar logisch duidelijker. In de C++ versie moet je in gedachten het programma uitvoeren om die logische structuur te zien. In Haskell kun je niet anders dan het algoritme logisch te structureren. Dit maakt het niet eenvoudiger om code over te schrijven uit een boek, maar zorgt er wel voor dat de programmeur beter zal begrijpen wat hij schrijft.

4.2.4 Hoofding

Een ander fundamenteel verschil is de hoofding van een functie in Haskell en C++. Bij Haskell worden de types gescheiden van de naam van de argumenten van een functie. Zoals reeds gezegd heeft dat het voordeel dat je in veel gevallen de typesignatuur kunt weglaten en minder code nodig hebt om algemeen bruikbare code te schrijven. Zolang de naam van de argumenten goed en logisch gekozen is maakt het code ook sneller leesbaar dan in C++. Naarmate een functie meer argumenten heeft, kan het omgekeerde gebeuren. Als aan de naam van het argument niet duidelijk te zien is, wat het conceptueel is (een functie, een

Figuur 4.3: Code van **itereerTotEinde** in Haskell om na elke iteratiestap het grootste verschil in het domein te tonen.

```
itereerTotEinde verandering probleem = itereer f (f probleem) probleem
  where
    f = zetIteratieStap
    grootsteverschil eerste tweede
      = foldl1 max [abs (x!(i,j) - y!(i,j))| i <- [0..(xabound x)],j<-[0..(yabound x)]]
      where
        x = werkMatrix eerste
        y = werkMatrix tweede
    itereer f x vorige | (grootsteverschil x vorige) < verandering = return x
                       | otherwise = do print (grootsteverschil x vorige)
                                        itereer f (f x) x
```

lijst of een eenvoudige variabele) kan het zijn dat je in een lange typesignatuur moet zoeken om dit te weten te komen. Als er geen typesignatuur is, zul je de betekenis van de functie moeten proberen te begrijpen of de typechecker van de compiler gebruiken. Op dit vlak is C++ duidelijker. Eén blik op het hoofd van de functie vertelt je van welk type een variabele is.

4.2.5 Doorgeven

Het doorgeven van functies behoeft in Haskell geen uitleg meer. In C++ heb je drie mogelijkheden. Ofwel met een functie als pointer, ofwel met een klasse met een functie als methode van die klasse waarvan voor elke functie de klasse afgeleid moet worden, ofwel met een functie als template parameter. Geen van deze zijn zo elegant en eenvoudig als in Haskell.

4.2.6 Pure functies versus niet-pure functies

In C++ is er geen onderscheid tussen beide soorten. In Haskell is er een groot verschil in de manier waarop je ze gebruikt. Niet pure functies moeten altijd in een **do** constructie staan. Dat het mengen van niet-pure met pure functies in code in Haskell niet vanzelfsprekend is, blijkt uit hetvolgende. Stel dat je in de code van **itereerTotEinde** na elke iteratiestap het maximumverschil wil afdrukken op het scherm, zodat je kunt volgen hoe snel de convergentie gebeurt. De verschillen zitten in de subfunctie **itereer**. Omdat **print** van het type **IO** is, verandert automatisch elke functie die **print** gebruikt in een niet-pure functie. Het resultaat is dat het resultaattype van **itereer** verandert in **IO oorspronkelijkType**. Dit betekent dat je niet meer **= x** kunt schrijven in de eerste guard. **x** is een matrix en moet verpakt worden in een **IO**-type. Dit doe je met **return**. De rest van de code van deze functie verandert niet. Op de plaats waar **itereerTotEinde** gebruikt wordt (zie regel 168 op pagina 95) verandert het type van **einde** ook in **IO oorspronkelijkType** en dit betekent dat je **einde** niet meer als argument aan **toon** kunt meegeven. De definitie van **einde** na de **where** moet verplaatst worden naar het **do**-construct in regel 159 op pagina 95. Daar schrijf je

```
einde <- itereerTotEinde verandering tweede
toon "data-einde.dat" einde
```

in plaats van

```
toon "data-einde.dat" einde
```

Het veranderen van één functie in een niet pure functie voor een praktische reden kan repercussies hebben voor veel andere code. Stel dat je hetzelfde wil doen zoals bij de functie **itereerTotEinde** voor een functie die door veel andere functies gebruikt wordt, dan kan dat zeer vervelende gevolgen hebben. Zelfs al ben je hiermee vertrouwd, dan nog leiden dergelijke details de aandacht af van datgene waar je mee bezig wil zijn. In C++ heb je dit probleem niet.

4.3 Typesysteem

4.3.1 Beperkingen op types

Iets gelijkaardigs aan het typesysteem met beperkingen van Haskell is mogelijk in C++. Met behulp van templates kun je het algoritme van Euclides in figuur 3.6a op pagina 51 schrijven als in figuur 4.4b op pagina 75. Het nadeel is, dat templates geen systeem van beperkingen is. Zolang de methodes **norm** en **modulo** op de gegeven types bestaan, zal deze functie werken maar het legt geen zichtbaar formele beperkingen op aan het gebruikte type van deze functie. Je moet ook gebruik maken van *getters* en *setters* om waarden uit het object te halen en er terug in te plaatsen bij de operaties die je in die klasse wilt ondersteunen. Terwijl je in Haskell de mogelijkheden *op* een type legt plaats je in C++ de mogelijkheden *in* de klasse.

Een andere mogelijkheid met een **Euclid**-klasse is

```
Euclid ggd(Euclid x, Euclid y)
```

De rest van het programma blijft hetzelfde. Zo leg je wel een beperking op, maar je bent beperkt tot één type dat verpakt is in de klasse **Euclid**. Je kunt ook het type dat **Euclid** verpakt meegeven met een template parameter, maar dan verandert de lijn in

```
template <class Type> Euclid<Type> ggd(Euclid<Type> x, Euclid<Type> y)
```

Je hebt nu wel een oplossing die conceptueel dichter ligt bij de Haskell versie maar het leest niet. Als je verschillende beperkingen op het type wil leggen, wordt het geheel nog ingewikkelder en zo goed als onleesbaar. Dit algoritme zal nooit direct werken op ingebouwde datatypes als **int** en **double**. Hiervoor zou je 4.4c op pagina 75 kunnen gebruiken, maar dan verlies je weer het idee van beperkingen zoals in Haskell en kun je geen klassen meer gebruiken om alles in te stoppen. Het systeem in Haskell is natuurlijker en het idee erachter is duidelijk vervat in het typesysteem. Je moet het niet nabouwen zoals je in C++ moet doen. In C++ krijg je door het gebruik van templates ook foutmeldingen, die enorm veel informatie bevatten en het debuggen van code moeilijk maken. Foutmeldingen in geparametriseerde code krijg je ook pas bij het effectief gebruiken van de code, daar waar je in Haskell de meeste fouten direct krijgt.

4.3.2 Wiskundig denken over types

Het systeem met beperkingen werkt in twee richtingen in Haskell. Langs één kant kun je duidelijk zichtbare typebeperkingen opleggen. Langs de andere kant kun je de beperkingen door Haskell zelf laten afleiden. Je kunt een functie schrijven zonder de types op te geven en daarna het meest algemene type van die functie opvragen met de interactieve commandolijn. In C++ kun je een meest algemene versie schrijven met gebruik van templates, maar je kunt nooit opvragen wat nu eigenlijk de beperkingen zijn, die op de functie of methode liggen.

In het Haskell lattice rules programma is het type van **generateLattice** (zie regel 280 op pagina 115) volgens de typechecker **(DivisionType a b, Ring (Modulo a), Enum (Modulo a))** => **[Modulo a]** -> **Modulo a** -> **[[b]]**. Dit vertelt ons, dat we van een vector over een quotiëntring en een waarde uit een quotiëntring een lijst van vectoren over het resultaat van de deling krijgen. De elementen van die quotiëntring moeten tot **Enum** behoren en tot een ring behoren. Zolang aan deze voorwaarden voldaan is, is een resultaat gegarandeerd. Je kunt aan de hand van de code en de typechecker van Haskell bijna net zo goed redeneren over deze wiskundige ideeën als je in de wiskunde zou doen. Je kunt beperkingen opleggen die strenger zijn dan het meest algemene geval, en dan kun je in concreto zien wat het resultaat daarvan is. In C++ is zoiets ondenkbaar.

4.3.3 Typeconversie

De strengheid van het typesysteem van Haskell voor conversie zorgt voor minder programmeerfouten. Waar in C++ een aantal conversies automatisch gebeuren, bijvoorbeeld van **double** naar **int**, met of zonder waarschuwing van de compiler, moet je die in Haskell expliciet geven. Dit automatisch omzetten van het ene type naar het andere in C++ kan tot subtiele programmeerfouten leiden. Als je de temperatuur van een stuk staaf wilt discretiseren aan de hand van twee parameters, namelijk de lengte en de afstand tussen twee opeenvolgende discrete punten, dan zou je geneigd zijn volgende code te schrijven

```
double dx, lengte;
int aantalpunten;
cin >> dx;
cin >> lengte;
aantalpunten = (lengte/dx)+1;
```

Dit compileert, mits een waarschuwing voor de lijn waar je een **double** toekent aan een **int**. Deze code geeft bij **dx = 0.1** en **lengte = 1.0** niet 11 maar 10 als waarde voor de variabele **aantalpunten**. Wat is er aan de hand? De conversie van **double** naar **int** trunceert de waarde naar omlaag. Om dit te vermijden moet je expliciet afronden met **round** uit de bibliotheek **cmath**. Ditzelfde probleem manifesteert zich in ons diffusieprobleem op regel 155 op pagina 88.

Dit zou de ervaren C++ programmeur niet mogen overkomen. In Haskell compileert zoiets gewoon niet. Daar kun je geen fractioneel getal toekennen aan een waarde die geheel moet zijn. De conversie moet daar expliciet gebeuren. Zodoende wordt zelfs de beginnende programmeur verplicht om na te denken welke afronding hij daar wil, en de overeenkomstige functie te kiezen die hiervoor nodig is. Overgangen van het ene type naar het andere komen regelmatig voor in numerieke code, en worden best niet (semi-)automatisch beslist door de compiler.

4.3.4 Overloading

Overloading in Haskell is niet te verwarren met die in C++. In Haskell kun je alleen een functie of operator overloaden als die tot een typeklasse behoort. Het aantal argumenten en de signatuur is vastgelegd in de definitie van de typeklasse. Je kunt daar niet van afwijken. Door een instantiedeclaratie kun je een bepaald datatype tot die typeklasse laten behoren en in de instantiedeclaratie de te overloaden operator of functie vastleggen. Dit heeft een aantal consequenties.

Figuur 4.4: Overloading en typesysteem

(a) Overloaden van functies in Haskell

```
class OverLoad a b where
  functie:: a -> b -> a

instance OverLoad Double Int where
  functie x y = x+ (fromIntegral y)
```

(b) Algoritme van Euclides met templates in C++

```
template <class Type> Type ggd(Type x, Type y)
{
     if (y.norm() <= 0) return x
     else
     {
          return ggd(y,x.modulo(y));
     }
}
```

(c) Algoritme van Euclides met templates in C++ zonder gebruik van klassen

```
template <class Type> Type ggd(Type x, Type y)
{
     if (norm(y) <= 0) return x
     else
     {
          return ggd(y,modulo(x,y));
     }
}
```

In Haskell kun je niet overloaden op het aantal argumenten. De functie die men overloadt moet steeds hetzelfde aantal argumenten hebben. De *wiskundige* operatoren die al in de Haskell prelude vastgelegd zijn in de klassen **Num** en **Fractional** kunnen hierdoor alleen werken op argumenten van hetzelfde type. Je kunt twee dingen doen om dit te omzeilen. Ofwel definieer je je eigen specifieke operator en klasse. We doen dit bij de algebraïsche klassen (zie regel 22 op pagina 111) en de scalaire vermenigvuldiging **.*** (zie regel 236 op pagina 114). Ofwel werk je met weglating van deze klassen uit de prelude. In het laatste geval zul je de klassen moeten herdefiniëren, zodat deze operatoren wel een signatuur krijgen die toelaat om twee argumenten van een verschillend type te gebruiken. Een eenvoudige oplossing, zoals het in dit geval in C++ mogelijk is, kan niet in Haskell.

In het algemene geval van een functie of operator met twee argumenten, waar je de mogelijkheid wilt om twee types te mengen, krijg je figuur 4.4a op pagina 75. Voor elke nieuwe definitie van **func** moet een nieuwe instantiedeclaratie ingevoerd worden. Dit laat toe om te overloaden op het return type van een functie. Dit doen we bij **generateLattice** in het programma voor lattice rules (zie regel 280 op pagina 115). Daar bepaalt de tweede parameter van **DivisionType** (zie regel 242 op pagina 115) het returntype van **generateLattice** door het gebruik van de operator **</>**. Je kunt daar verschillende versies van **</>** geven op hetzelfde type met een verschillend resultaat. In C++ is dit alleen mogelijk door het systeem van overloading na te bootsen met behulp van klassen (zie pagina 47 e.v. in Veldhuizen [1999]).

De striktheid van het typesysteem in Haskell voor overloading dwingt de ontwikkelaar tot een goed uitgedacht design. Waar je in C++ eenvoudig voor een uitzondering vlug een nieuwe functie of methode toevoegt of de operator overloadt, kan het voorkomen dat je in Haskell heel wat code moet aanpassen. Als je voor één zo'n geval heel je design moet aanpassen is dit zeer vervelend. Als dit regelmatig voorkomt tijdens de ontwikkeling kan het ook een fundamentele designfout blootleggen en de programmeur dwingen die te herstellen. Het is een tweesnijdend mes.

4.3.5 Pointers, geheugenbeheer en ongeïnitialiseerde waarden

In C++ kan men niet om het gebruik van pointers heen. Je hebt onvermijdelijk vroeg of laat te maken met alle vervelende nadelen van het gebruik ervan. Het is niet moeilijk om per ongeluk variabelen ongeïnitialiseerd te laten. Beide problemen heb je in Haskell niet en je vermijdt zo een aantal veel voorkomende fouten in numerieke code.

Vooral problematisch bij fouten met pointers is de oorzaak vinden. Bij de operator **<<** die we onder andere overloaden om polynomen te tonen in de C++ versie van lattice rules, moet je altijd de stream **s** uit het eerste argument teruggeven op het eind van de functie. Zoiets vergeten is vlug gebeurd. In dat geval duiken plots segmentatiefouten op, maar slechts in de helft van de uitvoering van hetzelfde programma. Als je de **while**-lussen aanpast of op een andere manier schrijft, krijg je soms geen fouten en soms wel. Na 5 uur vruchteloos zoeken op de verkeerde plaats bleek, dat we het statement **return s;** vergeten waren. In dit geval lag de fout dan nog dicht bij de plaats waar er soms iets verkeerd liep in de uitvoering. In Haskell is een gelijkaardige fout ondenkbaar.

Ook het gebruik van STL containers is niet altijd evident, tenzij je grondig nadenkt over tijdelijken en niet tijdelijken. Als je in de code voor **geefCoefficienten** vanaf regel 271 op pagina 103 geen referentie teruggeeft naar de vector waar de coëfficiënten in opgeslagen zijn, kun je geen correct gebruik maken van iterators. De code vanaf regel 532 op pagina 107 is dan foutief, want de iterator wijst naar het begin van een vector die een tijdelijke is en de

volgende regel niet meer bestaat. In Haskell zijn dergelijke zorgen niet aan de orde.

4.4 Modellering

4.4.1 STL versus datatypes

Het bewerken van containers uit STL is niet zo handig als bewerkingen op recursieve datastructuren, zoals lijsten, in Haskell. Vergelijk de scalaire vermenigvuldiging uit beide versies van het latticerule probleem. De Haskell versie (zie regel 236 op pagina 114) is een one-liner. In C++ (zie regel 149 op pagina 101) kost het vier lijnen. Daarvan zijn er twee nodig, die op zich niets functioneels doen. In de lijn die de vermenigvuldiging uitvoert, is de vermenigvuldiging niet meer te herkennen. Die vermenigvuldiging wordt uitgevoerd door **multiplies** wat een functor is. Daarnaast ben je verplicht begin en einde van de vector op te geven en nog eens de beginiterator om het resultaat in te plaatsen. In principe zou je een functie als **fmap** in Haskell kunnen schrijven. Maar dan blijf je zitten met **bind1st(multiplies<Type>(),x)** waar je in Haskell **(x<*>)** schrijft. **bind1st** doet hetzelfde als currying, maar dan alleen voor functors met twee argumenten. De vermenigvuldiging moet verpakt worden in een functor. In dit geval kun je de standaard meegeleverde functor gebruiken. Als je verder wilt gaan dan de standaard meegeleverde functors, zul je zelf een functor moeten schrijven om de functie of operator die je wilt gebruiken te kunnen toepassen op de container in kwestie.

Als we kijken naar de operaties op polynomen in beide versies zie je dat je 12 lijnen nodig hebt voor de optelling in C++ (zie regel 289 op pagina 103) en drie lijnen in Haskell (zie regel 151 op pagina 113). In C++ moet je expliciet rekening houden met het feit, dat de vectoren die de coëfficiënten opslaan verschillend van grootte kunnen zijn en ze expliciet allebei even groot maken. Een oplossing waarbij we alleen het stuk optellen dat even groot is en het restant aan het resultaat van de optelling hangen zou nog langer zijn. In Haskell is direct veel duidelijker wat gebeurt, en is het ook veel sneller geschreven.

De vermenigvuldiging van polynomen op dezelfde manier implementeren zou vervelend zijn. We hebben dan ook gebruik gemaakt van een andere manier met behulp van indices (zie regel 332 op pagina 104). Dit maakt de code korter, maar geeft kans op fouten met indices. In de Haskell versie (zie regel 163 op pagina 113) kun je duidelijk het recursieve idee achter de vermenigvuldiging zien, en je hoeft geen gebruik van indices te maken.

De deling van twee polynomen in C++ (zie regel 351 op pagina 104) is bijna zo kort als de Haskell versie (zie regel 174 op pagina 113). Bij deze operatie moet je veel minder structuren construeren en deconstrueren. De C++ versie ligt bijgevolg veel dichter bij de Haskell versie. Bij de deling van twee polynomen die resulteert in een serie, maak je uitgebreid gebruik van dergelijke constructie en deconstructie. De Haskell versie (zie regel 255 op pagina 115) is dan ook veel korter en bondiger dan de C++ versie (zie regel 480 op pagina 106).

STL-operaties in C++ zijn duidelijk minder handig om dergelijke structuren te bewerken dan patternmatching en dataconstructie in Haskell. Ook randvoorwaarden laten zich duidelijker leesbaar inbrengen met patternmatching.

4.4.2 Datatypes versus klassen

De modellering van de Peano axioma's (zie sectie 3.2.3 op pagina 52) is in Haskell niet zo moeilijk door het gemak waarmee je nieuwe datatypes kunt definiëren en die vervolgens kunt hanteren met pattern matching. Hetzelfde in C++ doen wordt een ingewikkelde en

onoverzichtelijke programmeeroefening. Datatypes die bestaan uit wat verschillende waarden van verschillende bestaande types kun je verpakken in klassen. Als het gaat om nieuwe types, dan moet je een representatie uitwerken met behulp van de bestaande types van C++. In C++ kun je evenwel niet zomaar recursieve types definiëren. Je moet ze simuleren en dat zal werk en tijd kosten, wat helemaal niet nodig is in Haskell.

Omdat nieuwe types met speciale eigenschappen regelmatig opduiken in de wiskunde, is dit een duidelijk voordeel van Haskell. Je kunt ze eenvoudig definiëren. Je kunt ze instanties maken van bestaande klassen om er de *natuurlijke* bewerkingen op vast te leggen. Je kunt nieuwe typeklassen aanmaken om eigenschappen van die nieuwe types te abstraheren. Je kunt ze eenvoudig schrijven en herkennen in code, daar waar je in C++ telkens eerst een object moet aanmaken voor je het kunt gebruiken. Je kunt ze eenvoudig op het scherm tonen door Haskell zelf de instantie van de klasse **Show** te laten afleiden met **deriving**. In C++ moet je dit altijd zelf doen. Door pattern matching, zal je geen *getters* en *setters* moeten schrijven per nieuw type. Bij grote datatypes, waar je veldlabels gebruikt in plaats van pattern matching hoef je de getters en setters niet eens zelf te schrijven.

C++ is dan weer handiger als het gedrag van een type complexer wordt. Datatypes in Haskell laten niet toe om bij het schrijven van een bepaald element van dat type automatisch een aantal bewerkingen uit te voeren. Hiervoor moet je een afzonderlijke functie schrijven die telkens toegepast moet worden op dat element. Die zal op de plaats komen waar we een element van dat type schrijven. Bij het **Modulo**-type bij lattice rules in Haskell zou het handig zijn om op het modulonummer direct de modulo operatie toe te passen, zodat het modulonummer altijd *kleiner* is dan het modulogetal. In C++ doe je dit bij de creatie van het object. In Haskell is er altijd een strikte scheiding tussen de data, bijvoorbeeld het datatype **diffusion2d** (zie regel 27 op pagina 93) bij het diffusieprobleem, en de implementatie die die data gebruikt. In C++ zitten data en implementatie samen in één klasse en heb je controle over hoe data gecreëerd wordt. Dit is eerder geschikt om een probleem te lijf te gaan, zoals bij het diffusieprobleem, maar niet zozeer om relatief eenvoudige types te definiëren, zoals bij lattice rules.

4.4.3 Lazy evaluation

Dit concept bestaat in C++ niet. Zoals in sectie 3.3 op pagina 53 al uitvoerig belicht is, is dit een zeer krachtig mechanisme, waarmee je eenvoudig om kunt gaan met wiskundige structuren die potentieel oneindig zijn.

4.5 Debuggen

4.5.1 Informatie tijdens de uitvoering tonen

Een gelijkaardige stijl van debuggen met **printf** of **cout** in C++ is ook mogelijk in Haskell. In Haskell kun je hiervoor **trace** uit de module **Debug.trace** gebruiken. **trace** kun je overal waar een uitdrukking staat plaatsen. Het eerste argument is een string die getoond wordt waarna de uitdrukking in het tweede argument geëvalueerd wordt. Meestal wil je op het scherm tonen wat die uitdrukking is. Dit betekent veelal dat je ofwel de uitdrukking volledig moet herhalen in het eerste argument van **trace**, ofwel moet je speciaal hiervoor de uitdrukking achter een **where** plaatsen en de uitdrukking daar toekennen aan een kortere naam. Als je in code die een iteratieschema toepast de gegenereerde waarden wilt dumpen op

het scherm, dan zou de code in figuur 4.5a op pagina 80 ofwel moeten veranderen in figuur 4.5b of in figuur 4.5c. Je kunt hier niet

```
bereken i j | hoortTotDomein (i,j) = trace (show (bereken i j))
                                       t!(i,j) + c*(t!(i-1,j) -4*t!(i,j)
                                       + t!(i+1,j) + t!(i,j-1) +t!(i,j+1))
```

schrijven want Haskell belandt zo in een oneindige lus. In gelijkaardige C++ code zul je de uitdrukking toegekend hebben aan een variabele en kun je in dit geval het volgende schrijven.

```
cout << t(i,j);
```

Dit is veel eenvoudiger.

Het debuggen met **trace** kan ook vervelend worden als je meer informatie op het scherm wilt. Stel dat je op dezelfde plaats ook de indices op het scherm wilt, dan zal in de versie in figuur 4.5c het trace statement veranderen in

```
trace (show i ++ " " ++ show j ++ " " ++ show res) res
```

In C++ kun je één van volgende statements schrijven

```
cout << i << " " << j << " " << t(i,j);
printf("%i %i %f",i,j,t(i,j));
```

In Haskell ben je verplicht om elke niet-string om te zetten met **show** en alle strings te bundelen tot één string met **++**.

4.5.2 Tellen

Als je wil nagaan hoeveel keer **bereken** opgeroepen wordt, kun je in C++ een globale variabele of klassevariabele declareren, en die ergens in de code van **bereken** plaatsen en verhogen telkens bereken opgeroepen wordt. In Haskell bestaan er geen globale variabelen noch data hiding. Daar kun je niet anders dan ofwel die variabele door te geven met de functie-aanroep, ofwel van alle functies niet pure functies maken en een substituut van een globale variabele in Haskell met monads creëren. Monads zijn niet-pure functie met alle vervelende gevolgen die we al in sectie 4.2.6 op pagina 72 besproken hebben. Een dergelijke *oplossing* ziet er als figuur 4.5d op pagina 80 uit. Hierbij is **f** de functie waarvan je wil tellen hoeveel keer ze opgeroepen wordt. **inc** is de functie die het tellen uitvoert. Met **runCounted** kun je de functie laten evalueren. **runCounted** geeft een paar terug met als eerste element het resultaat na **f** en als tweede de teller. Je bent hier verplicht om de functie **f** volledig sequentieel in een **do** statement te schrijven, omdat **f** een monad en dus een niet-pure functie is. Dit zal ook gelden voor alle functies die **f** aanroepen. Dit doet zo goed als alle voordelen van functionele talen teniet.

Beide voorgestelde oplossingen, namelijk monads of het doorgeven van de waarde met het resultaat van de functie, zijn elegant noch eenvoudig. Ze vereisen het aanpassen van code op veel plaatsen. Dit is in gelijkaardige code in C++ niet nodig. Een andere mogelijkheid is, om aan de hand van een profiler na te gaan hoeveel keer de functie in kwestie geëvalueerd wordt. Dit dwingt tot compileren met profilerinformatie en een analyse van het resultaat van de uitvoering. Deze problemen zijn inherent aan een taal met een dergelijk strikt typesysteem en is duidelijk een praktisch minpunt voor ontwikkeling.

Figuur 4.5: Debuggen in Haskell

(a) Iteratieschema zonder debug statement.

```
bereken i j | hoortTotDomein (i,j) = t!(i,j) + c*(t!(i-1,j) + t!(i+1,j)
                                      + t!(i,j-1) +t!(i,j+1)-4*t!(i,j))
            | otherwise = t!(i,j)
```

(b) Iteratieschema met debug statement

```
bereken i j | hoortTotDomein (i,j) = trace
                                     (show (t!(i,j) + c*(o!(i-1,j) + t!(i+1,j)
                                     + o!(i,j-1) +t!(i,j+1) - 4*t!(i,j))))
                                     t!(i,j) + c*(o!(i-1,j) + t!(i+1,j)
                                     + o!(i,j-1) + t!(i,j+1) - 4*t!(i,j))
            | otherwise = t!(i,j)
```

(c) Iteratieschema met debug statement

```
bereken i j | hoortTotDomein (i,j) = trace (show res) res
            | otherwise = t!(i,j)
              where
                res = t!(i,j) + c*(t!(i-1,j) -4*t!(i,j)
              + t!(i+1,j) + t!(i,j-1) +t!(i,j+1))
```

(d) Een monad teller in Haskell

```
import Control.Monad.Writer

newtype Counter = Counter Integer deriving (Eq, Show, Num, Ord, Enum)
instance Monoid Counter where mempty = Counter 0; mappend = (+)
newtype Counted a = Counted (Writer Counter a)
                    deriving (Functor, Monad, MonadWriter Counter)

runCounted (Counted w) = runWriter w

inc :: Counted ()
inc = tell 1

f 0 = return 0
f 1 = return 1
f n = do
    inc
    x <- f (n-1)
    y <- f (n-2)
    return (x+y)
```

(e) Dezelfde functie zonder gebruik van monads.

```
f 0 = 0
f 1 = 1
f n = (f (n-1)) + (f (n-2))
```

4.6 Conclusie

Haskell heeft de numerieke ontwikkelaar heel wat te bieden. Een functionele taal ligt qua schrijfwijze dichter bij een wiskundige schrijf- en denkwijze. Het functieconcept zelf staat qua gebruik zeer dicht bij het wiskundige equivalent. Complex gestructureerde wiskundige objecten laten zich snel schrijven en bewerken in Haskell. Algemeen bruikbare functies schrijven kost minder werk dan in C++, waar het juist extra werk vraagt. Typeconversie wordt nooit aan de compiler overgelaten. Lambdafuncties geven de mogelijkheid om snel eenvoudige functies te schrijven en aan te passen. Ongeïnitialiseerde waarden, pointers en de bijbehorende geheugenproblemen zijn onbestaande. Het typesysteem laat door het gebruik van typeklassen modellering toe van wiskundige concepten die je in C++ helemaal niet kunt implementeren of uitdrukken. Het strikte typesysteem dwingt de programmeur tot discipline. Datatypes laten snel toe eenvoudige wiskundige structuren te modelleren. Lazy evaluation vereenvoudigt het omgaan met structuren, die geen vast omlijnde grenzen hebben. Tot slot heeft elke Haskell-compiler een interactieve commandoversie waarop je functies en volledige programma's kunt uittesten met verschillende argumenten, zonder dat je telkens het programma moet aanpassen, hercompileren en uitvoeren. Om eenzelfde flexibiliteit in C++ te bereiken, moet je heel wat extra programmeerwerk doen.

C++ heeft dan weer een aantal voordelen ten opzichte van Haskell. Programma's zijn veel sneller dan hun Haskell-equivalenten. De ontwerpmogelijkheden van C++ zijn reeds lang en uitvoerig in de praktijk gebracht en beschreven. Er is al heel wat numerieke code in C++ geschreven en er is ook al heel wat over gepubliceerd. In Haskell sta je regelmatig alleen en moet je dingen zelf uitvinden. C++ heeft zijn nut al bewezen in grootschalige projecten, daar waar dit voor Haskell nog niet het geval is. Er zijn ook heel wat numerieke bibliotheken beschikbaar in C++. In Haskell zijn die zo goed als onbestaande. Je kunt ze wel gebruiken aan de hand van de Foreign Function Interface (FFI). Dit is evenwel niet vanzelfsprekend. Haskell is lang niet zo uitgebreid gedocumenteerd als C++. Ook het debuggen van Haskell-programma's is niet zo eenvoudig als dat van C++-programma's.

Het is duidelijk dat je Haskell niet gebruikt voor projecten waar je zware number-crunching moet doen. Hiervoor gebruik je C++. Als je veel moet modelleren en je hebt geen ultra-efficiënt programma nodig, dan is Haskell zeker te verkiezen boven C++. De twee problemen die wij geïmplementeerd hebben tonen dit duidelijk aan. Toch is het opvallend dat er zelfs in rekenintensieve toepassingen veel te zeggen is voor Haskell, en dan vooral voor de manier waarop je wiskundige concepten kunt uitdrukken. Ideaal zou een taal zijn die de snelheid en grootschalige ontwerpmogelijkheden van C++ combineert met het functionele karakter van Haskell. Dit lijkt het doel te zijn van een project dat momenteel bij Sun loopt, en geleid wordt door één van de medeontwerpers van Java. Het project ontwikkelt momenteel een standaard voor een nieuwe taal Fortress (Steel [2005]), die specifiek bedoeld is voor numeriek ontwerp. De taal bevindt zich momenteel nog in het stadium van ontwerp. Ondanks het feit dat de ontwerpers een aantal functionele talen als inspiratiebron aanhalen, is het nog maar de vraag hoeveel daarvan uiteindelijk in de standaard van Fortress terecht zal komen. Wij konden in het huidige ontwerp (Allen et al. [2005]) alvast niet veel *functioneels* vinden.

Hoofdstuk 5

Besluit

Voor het volledige eindwerk heb ik gebruik gemaakt van de standaarden van de talen die ik bespreek. Voor Fortran is dat JTC1/SC22 [2004]. Voor C++ hadden wij niet de beschikking over de standaard, maar wel over Stroustrup [2000] en de C standaard ISO [1999] die ook van toepassing is. Voor Haskell bestaat de standaard uit 1998 (Jones [2002]), maar die wordt continu aangevuld in de implementaties van de twee belangrijkste Haskell compilers Hugs en vooral GHC. Ik heb dan ook regelmatig beroep gedaan op de documentatie van GHC (GHC [2005]).

Het eerste hoofdstuk geeft een beschrijving van de faciliteiten die de IEEE 754 standaard biedt, hun mogelijke nut voor numerieke toepassingen en een opsomming en/of vergelijking van wat de talen bieden. Daarin is telkens de bron geciteerd, als ik gebruik maak van informatie die ik niet uit de standaard gehaald heb. Dit hoofdstuk is een samenvatting en vergelijking van de gevonden informatie.

Voor het tweede hoofdstuk over objectgericht programmeren, heb ik gebruik gemaakt van volgende informatie naast Stroustrup [2000]: Veldhuizen [1999], Bulka and Mayhew [1999] en Goldthwaite [2003]. Waar ik gebruik gemaakt heb van andere informatie, is die in de tekst geciteerd. Ook dit hoofdstuk is grotendeels een samenvatting van het gevonden materiaal. In sectie 2.4.2 op pagina 28 over expression templates, heb ik de C++ code zelf ontwikkeld. Die kunt u ook vinden op de webpagina `http://www.reinaert.albrecht.easynet.be/ithesis/`. Wat het stukje over Java aangaat, heb ik alle bronnen in de tekst geciteerd.

Voor het derde hoofdstuk over functioneel programmeren heb ik naast de standaard en de documentatie van GHC ook dankbaar gebruik gemaakt van de Haskell referentie in Miloslav [2006]. De ideeën uit de twee papers (die over groepentheorie en lazy evaluation), die ik voor dit hoofdstuk gebruik, heb ik niet zelf geproduceerd. Voor de rest heb ik zelfstandig de relevantie voor numerieke software van de concepten, die in Haskell bestaan, uitgedacht. Uiteraard hebben de goede suggesties en opmerkingen van mijn assistenten ook een rol gespeeld.

De implementatie van de twee problemen is volledig van mijn hand, maar zou onmogelijk geweest zijn zonder de talrijke aanmerkingen van mijn assistenten die steeds mijn aandacht vestigden op denkfouten. Zij hebben mij ook de problemen aan de hand gedaan. De vergelijking van C++ met Haskell is volledig van mijn hand. Bij de implementatie van de problemen in Haskell heb ik ook veel implementatiesuggesties gekregen van de talrijke enthousiaste aanwezigen in het **#Haskell**-kanaal op het Freenode IRC-netwerk (Peer-Directed Projects Center [2006]). Zij hebben mij telkens veel bijgebracht over hoe Haskell werkt, als

ik onmogelijke dingen wou proberen in Haskell.

Bij het begin van dit eindwerk was er niet echt een doelstelling, omdat niemand ooit een dergelijke studie bleek ondernomen te hebben. De mogelijkheden waren zeer breed. Pas gaandeweg zijn de doelstellingen gegroeid. De eerste twee hoofdstukken (over de IEEE 754 standaard en numeriek objectgericht ontwerp) zijn een direct resultaat van de vaststelling, dat een dergelijke *opsomming* nooit gebeurd was. Lange tijd wist ik niet dat de volledige IEEE 754 standaard in de nieuwe Fortran standaard opgenomen was, en ook geïmplementeerd was in de NAG Fortran compiler. De informatie buiten de standaard over Fortran bleek zeer schaars, en ik heb veel tijd verloren met informatie te zoeken die onvolledig en niet meer up to date is.

Van het tweede hoofdstuk vind ik dat ik niet genoeg in detail ben kunnen gaan. De mogelijkheden die ik in functionele talen zag, leken mij interessanter om uit te werken dan nog maar eens iets over C++ te schrijven, waar toch al veel over gepubliceerd was. Grote lacunes zijn hier zeker dat ik niets aangaande template metaprogramming en traits heb opgenomen.

Voor het functionele hoofdstuk was de oorspronkelijk doelstelling, om alleen wat voorbeelden te geven van hoe een functionele taal andere problemen kan aanpakken, dan met een imperatieve taal mogelijk of praktisch is. Voor dit hoofdstuk heb ik heel veel tijd verloren met het lezen van papers over quantumcomputersimulatie waar ik niet veel van begreep. Ik heb er ook niet veel mee kunnen doen voor mijn eindwerk. Die tijd had beter kunnen gespendeerd worden. Er bleek genoeg tijd te zijn om systematisch alle kenmerken van Haskell te bekijken en te zien of die het schrijven van numerieke code vereenvoudigen.

De doelstelling voor het laatste hoofdstuk was om een grondige vergelijking te maken van C++ en Haskell. Dit is mijns inziens geen ideale vergelijking geworden omwille van een aantal redenen. Ten eerste vergelijk je twee talen die op een fundamenteel andere manier problemen oplossen. Het denken in deze talen is compleet verschillend. Een vergelijking tussen beide is sowieso geen eenvoudige opdracht. Om een dergelijke vergelijking echt tot een goed einde te brengen, zou ik idealiter zo vroeg mogelijk hiermee moeten begonnen zijn. Dit is niet gebeurd, doordat ik hals over kop de problemen ben beginnen implementeren zonder eerst een grondige analyse en ontwerp van de problemen te maken. Ik heb dan ook zo goed als alle fouten gemaakt die bij het implementeren van deze problemen mogelijk zijn. Daar heb ik wel veel uit geleerd maar het resultaat was wel dat ik pas heel laat de vergelijking tussen beide programmeertalen aangevat heb. De vergelijking blijft mijns inziens een beetje teveel steken op een laag niveau.

Indien ik mijn eindwerk had kunnen starten met het inzicht dat ik nu heb zou ik begonnen zijn met het implementeren van een aantal problemen in de twee talen die ik gekozen heb om zoveel mogelijk praktische ervaring op te doen met het implementeren van numerieke problemen in beide talen.

De belangrijkste conclusie uit mijn eindwerk is dat Haskell verrassend veel mogelijkheden biedt om het programmeren van numerieke toepassingen te vereenvoudigen. Tot slot waag ik mij nog aan een onvoorzichtige voorspelling: als quantumcomputers ooit algemeen gebruikt worden zou het kunnen zijn dat in de nabije of verre toekomst hoofdzakelijk op een functionele manier geprogrammeerd zal worden. Maar iedereen weet hoe voorzichtig je moet zijn met toekomstvoorspellingen...

Bijlage A

Code

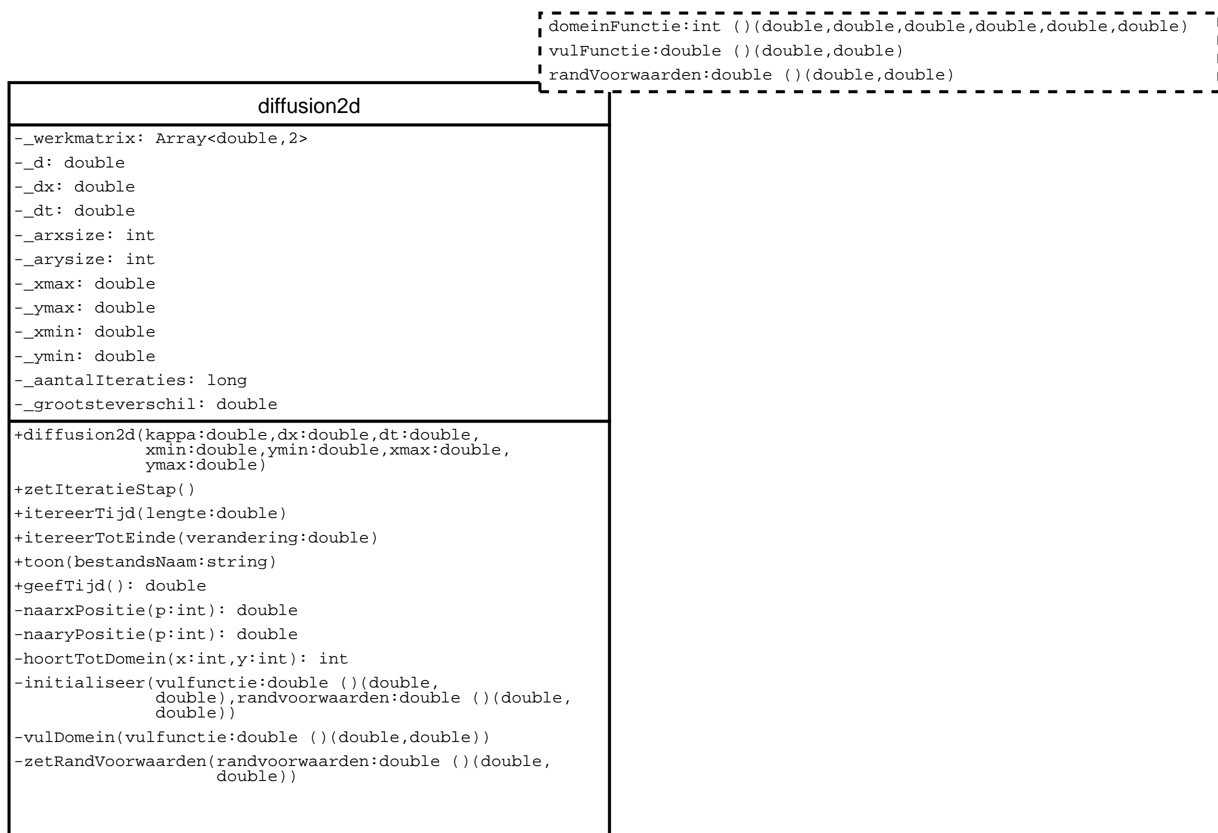
Alle code is ook online te vinden op `http:\\www.reinaert.albrecht.easynet.be\ithesis\` samen met een aantal andere stukken code die tijdens het schrijven van dit eindwerk tot stand kwamen.

A.1 Diffusieprobleem

A.1.1 C++ versie

```
#include <blitz/array.h>
#include <fstream>
#include <string>
#include <cmath>
#include "tijd.h"

using namespace blitz;

/* diffusion2d bevat alle methodes om een diffusievgl. van de vorm
 *
 * du/dt = d^2u/dx^2 + d^2u/dy^2
 *
 * op te lossen door discretisatie.
 *
 * Het gebruikte iteratieschema is:
 *   T_n+1(i,j) = T_n(i,j)+ C( T_n+1(i-1,j) -2T_n(i,j)+T_n(i+1,j)
 *                   + T_n+1(i,j-1)-2T_n(i,j)+T_n(i,j+1))
 *
 * waarbij C gelijk is aan dt/dx^2
 *
 * template parameters:
 * -> domeinFunctie
 * bepaalt of een gegeven positie op het domein ligt of
 * niet
 * de eerste twee parameters zijn de positie in het domein
 * de laatste vier parameters zijn de afmetingen van de werkmatrix
 * -> vulFunctie
 * vult het domein op in functie van de positie in het domein
 * -> randVoorwaarden
 * bepaalt de randvoorwaarden in functie van de positie buiten het domein
 */
```

Figuur A.1: Klassendiagram van het diffusieprobleem



Tabel A.1: Overzicht van de methodes in de C++ klasse **diffusion2d**

Methodenaam	
zetIteratieStap	Zet één iteratiestap.
itereerTotEinde	Zet iteratiestappen totdat het het grootste verschil in het domein kleiner is dan de opgegeven waarde.
itereerTijd	Evolueer het probleem gedurende de opgegeven simulatietijd.
geefTijd	Geef het aantal seconden dat gesimuleerd is sinds het begin van de simulatie.
toon	Plaatst de waarden uit de matrix samen met hun werkelijke positie in het bestand met de naam die opgegeven is als parameter.
naarxPositie	
naaryPositie	Zet een index in de matrix om in zijn werkelijke positie in het domein.
hoortTotDomein	Geeft terug of de opgegeven positie in de matrix tot het domein van het probleem hoort.
initialiseer	Vult het domein en zet de randvoorwaarden met de functies opgegeven als template parameter.
vulDomein	Vult het domein met waarde aan de hand van de functie opgegeven als template parameter.
zetRandVoorwaarden	Zet de randvoorwaarden aan de hand van de functie opgegeven als template parameter.

```
template <int (domeinFunctie)(double,double,double,double,double,double),
      double (vulFunctie)(double,double),
35    double (randVoorwaarden)(double,double)>
      class diffusion2d
{
    // de array van alle waarden die al of niet in het domein liggen
    Array<double,2> * _werkmatrix;
40  // de constante D
    double _d;
    // de afstand tussen twee opeenvolgende punten zowel in de x als y
    // richting
    double _dx;
45  // de afstand in seconden tussen twee tijdsstappen
    double _dt;
    // de hoogste x en y indices van _werkmatrix
    int _arxsize, _arysize;
    // de grootte van het domein in de x als y richting
50  // het domein gaat altijd van 0 naar _xsize in de x-richting
    // en 0 naar _ysize in de y-richting
    double _xmax, _ymax, _xmin, _ymin;
    // aantal iteraties
    long _aantalIteraties;
55
    // maximum verschil tussen twee domeinwaarden uit de iteratie die
    // voorbij is en de domeinwaarden voor de iteratie
    double _grootsteverschil;
```

```
60
        // zet een index om naar de werkelijke positie in het domein
        double naarxPositie(const int &p) const
        {
            return p*_dx+_xmin;
65      }
        double naaryPositie(const int &p) const
        {
            return p*_dx+_ymin;
        }
70
        /**
         * controleert of de gegeven indices tot het domein behoren of niet.
         * Deze methode sluit uit dat de gebruiker accidenteel een
         * functie opgeeft die de randen van _werkmatrix ook als
75       * domeinwaarden zou beschouwen en zo de methode zetIteratieStap
         * over de grenzen van _werkArray laat gaan.
         */
        int hoortTotDomein(const int &x, const int &y)
        {
80          if ((x <= 0) || (y <= 0) || (x >= _arxsize-1) || (y >= _arysize-1)) return 0;
            else if (domeinFunctie(naarxPositie(x),naaryPositie(y),_xmin,_ymin,_xmax,_ymax))
                return 1;
            else return 0;
        }
85
        /**
         * Vult het domein en zet de randvoorwaarden met de gegeven
         * functies
         */
90      template <double (vulfunctie)(double,double),
             double (randvoorwaarden)(double,double)>
            void initialiseer()
        {
            vulDomein<vulfunctie>();
95          zetRandVoorwaarden<randvoorwaarden>();

        }

        /**
100      * Vult het domein met waarden gegeven door de opgegeven functie
         */
        template <double (vulfunctie)(double,double)> void vulDomein()
        {
            Array<double,2>::iterator it = (*_werkmatrix).begin();
105
            TinyVector<int,2> pos;
            int x,y;
            do
            {
110             pos = it.position();
                x = pos(0);
                y = pos(1);
                if(hoortTotDomein(x,y)) *it = vulfunctie(naarxPositie(x),naaryPositie(y));

115         }while (++it != (*_werkmatrix).end());
        }
```

```
    /**
     * Zet de randvoorwaarden aan de hand van de opgegeven functie
     *
     * De randvoorwaarden hoeven slechts eenmaal gezet te worden
     * voor de eerste iteratiestap gezet wordt.
     * Die worden door iteratiestappen niet gewijzigd.
     */
    template <double (randvoorwaarden)(double,double)> void zetRandVoorwaarden()
    {
        Array<double,2>::iterator it = (*_werkmatrix).begin();

        TinyVector<int,2> pos;
        int x,y;
        do
        {
            pos = it.position();
            //cout << "rand " << pos(0) << " " << pos(1) << "\n";
            x = pos(0);
            y = pos(1);
            if(!hoortTotDomein(x,y))
                (*it) = randvoorwaarden(naarxPositie(x),naaryPositie(y));

        }while (++it != (*_werkmatrix).end());

    }
    public:

    /**
     * Maakt een nieuw diffusieprobleem
     *
     * kappa: de diffusieconstante
     * dx: afstand tussen twee opeenvolgende punten in het domein
     * dt: afstand in tijd tussen twee iteratiestappen
     * xsize en ysize: grootte van de werkmatrix
     */
    diffusion2d(double kappa,double dx, double dt,double xmin, double ymin,
            double xmax, double ymax)
        :_dx(dx),_dt(dt),_d((kappa*dt)/(dx*dx)), _arxsize((int)round((xmax-xmin)/dx+1)),
        _arysize((int)round((ymax-ymin)/dx+1)),_xmin(xmin),_ymin(ymin),
        _xmax(xmax),_ymax(ymax),_grootsteverschil(0.0),_aantalIteraties(0)
    {
        _werkmatrix= new Array<double,2>(_arxsize,_arysize);
        (*_werkmatrix) = 0; // initialiseer op nul
        // vul het domein en zet de randvoorwaarden
        initialiseer<vulFunctie,randVoorwaarden>();
    }

    /**
     * Zet een iteratiestap
     */
    void zetIteratieStap()
    {
        Array<double,2> _nieuwematrix(_arxsize,_arysize);

        TinyVector<int,2> pos;
        double oudeWaarde;
```

```
        int x,y;
        _aantalIteraties++;
        // zet _grootsteverschil op 0
        _grootsteverschil = 0.0;
        for (x=0; x <_arxsize; x++)
            for(y = 0; y < _arysize; y++)
        {
                // vraag de positie van het huidige punt op
            if(hoortTotDomein(x,y))
            {
                // sla de oude waarde van het domeinpunt op
                oudeWaarde = (*_werkmatrix)(x,y);
                // pas iteratieschema toe
                _nieuwematrix(x,y)
                    = (*_werkmatrix)(x,y)
                    + _d*((_nieuwematrix)(x-1,y) + (*_werkmatrix)(x+1,y)
                    + (_nieuwematrix)(x,y-1) + (*_werkmatrix)(x,y+1)
                    - 4*(*_werkmatrix)(x,y));
                // controleer of de nieuwe waarde een groter verschil
                // heeft met de oude waarde dan in _grootsteverschil
                // opgeslaan is
                if(fabs(oudeWaarde - (_nieuwematrix(x,y))) > _grootsteverschil)
                    _grootsteverschil = fabs(oudeWaarde - (_nieuwematrix(x,y)));
            }
            else _nieuwematrix(x,y) = (*_werkmatrix)(x,y);

        }

        *_werkmatrix = _nieuwematrix;
    }


    /**
     * Plaats de waarden uit werkmatrix in een bestand gegeven door bestandsnaam
     * zodat gnuplot er een 3d grafiek mee kan plotten
     */
    void toon(const string &bestandsNaam)
    {
        ofstream bestand;
        bestand.open(bestandsNaam.c_str());
        Array<double,2>::iterator it = (*_werkmatrix).begin();
        // zet de precisie op het maximum
        bestand.precision(16);

        TinyVector<int,2> pos;
        double x,y;
        do
        {
            pos = it.position();
            bestand << naarxPositie(pos(0)) << " "
                << naaryPositie(pos(1)) << " "
                << (*it) << "\n";
            // scheid elk blok af met een newline
            if (pos(1) == _arysize-1) bestand << "\n";
        } while (++it != (*_werkmatrix).end());
        bestand.close();
    }
```

```
        /**
         * Zet iteratiestappen tot de grootste verandering in het domein
235      * tot de vorige iteratiestap kleiner is den verandering
         * verandering: de kleinste verandering die nodig is in het domein
         *              tussen twee opeenvolgende iteratiestappen zodat
         *              iterereerTotEinde verder blijft lopen. Is het
         *              maximum verschil verschil tussen twee punten uit
240      *              het domein kleiner dan stopt itereerTotEinde
        */
        void itereerTotEinde(double verandering)
        {
            while (_grootsteverschil >= verandering)
245         {
                zetIteratieStap();
            }
        }

250     /**
         * Zet iteratiestappen tot het gegeven aantal seconden gesimuleerd is.
         *
         * seconden: tijd in seconden
         */
255     void itereerTijd(double seconden)
        {
            long iteraties = (long)round(seconden/_dt);
            for (long i=0; i < iteraties; i++)  zetIteratieStap();
        }
260
        /**
         * Geef de tijd die verlopen is in het probleem door het zetten van
         * iteratiestappen
         */
265     double geefTijd()
        {
            return _aantalIteraties * _dt;
        }

270 };


    /**
     * Functie die gebruikt kan worden om een cirkelvormig domein te bepalen
275  */
    int domainCircle(double x,double y, double Xmin, double Ymin, double Xmax, double Ymax)
    {
        if (sqrt(pow(x,2.0) + pow(y,2.0)) < (Xmax/10)) return 0;
        else    if (sqrt(pow(x,2.0) + pow(y,2.0)) < (Xmax)) return 1;
280     else return 0;
    }

    /**
     * Een functie die gebruikt kan worden het domein op te vullen met waarden
285  */
    double vulFunctie(double x, double y)
    {
```

```
    return 0;
}
290
/**
 * Een functie die gebruikt kan worden om randvoorwaarden te zetten
 */
double randVoorwaarden(double x,double y)
295 {
    if (sqrt(pow(x,2.0) + pow(y,2.0)) < (0.1)) return 20;
    return 10;
}

300 int main ()
{
    double dx,verandering,kappa;
    double xmin,ymin,xmax,ymax;
    double tijda, tijdb;
305 // lees parameters in
    cin >> kappa ;
    cin >> dx;
    cin >> verandering;
    cin >> tijda;
310 cin >> tijdb;
    cin >> xmin;
    cin >> xmax;
    ymin = xmin;
    ymax = xmax;
315 // zet het probleem op
    diffusion2d<domainCircle,vulFunctie,randVoorwaarden>
        probleem(kappa,dx,0.001,xmin,ymin,xmax,ymax);
    // plaats de startwaarden in een bestand
    probleem.toon("data-start.dat");
320 // laat de iteratiestappen tijda lopen
    probleem.itereerTijd(tijda);
    // plaats de gegevens in een bestand
    probleem.toon("data-tijda.dat");
    // laat de iteratiestappen nog tijdb-tijda lopen
325 probleem.itereerTijd(tijdb-tijda);
    // plaats de gegevens in een bestand
    probleem.toon("data-tijdb.dat");
    // itereer tot het probleem geconvergeerd is
    probleem.itereerTotEinde(verandering);
330 // plaats de gegevens in een bestand
    probleem.toon("data-einde.dat");
    // toon de verstreken tijd sinds het begin van de simulatie
    cout << probleem.geefTijd() << "\n";
}
```

A.1.2 Haskell versie

```
{-# OPTIONS #-}
{- Caveat: gebruikt 98 extensies van haskell. Werkt om niet
met hugs. Gebruik GHC.

5 compileren:

ghc -O2 diffusion2d.hs -o naamuitvoerbaarbestand
-}
```

Tabel A.2: Overzicht van de functies in de Haskell versie van het diffusieprobleem

Functienaam	
xbound **ybound**	Geeft de grootste index in de x- of y-richting van het gegeven diffusieprobleem.
xabound **yabound**	Geeft de grootste index van de gegeven matrix in x- of y-richting.
naarPositie	Geeft de werkelijke positie van de opgegeven indices in de matrix.
hoortTotDomein	Geeft terug of de opgegeven positie in de matrix tot het domein van het probleem hoort.
zetIteratieStap	Zet één iteratiestap.
geefMatrix	Maak een matrix en vul het domein met waarden van de opgegeven domeinfunctie en zet de randvoorwaarden aan de hand van de opgegeven randfunctie.
domeinCirkel	Definieert een cirkelvormig domein.
toonArray	Zet een gegeven array om in een string die gebruikt kan worden door **toon** om de waarden uit de matrix in een bestand te plaatsen.
toon	Plaatst de waarden uit de matrix samen met hun werkelijke positie in het bestand met de naam die opgegeven is als parameter.
itereerTotEinde	Zet iteratiestappen totdat het het grootste verschil in het domein kleiner is dan de opgegeven waarde.
itereerTijd	Evolueer het probleem gedurende de opgegeven simulatietijd.
geefTijd	Geef het aantal seconden dat gesimuleerd is sinds het begin van de simulatie van het probleem.

```
module Main where
import System
import Numeric
import Data.Array.Diff
import Debug.Trace
import IO
import CPUTime
import Data.List

{-
Datastructuur die de verschillend parameters van een diffusieprobleem bevat:
- grootte van het domein
- ruimtediscretisatie
- tijdsdiscretisatie
- de functie die bepaalt welke waarden tot het domein behoren
- het aantal iteraties dat gezet is met het probleem
- de matrix waarin domeinwaarden en randwaarden liggen
-}
data Diffusie2d a b i = Diff2d {kappa,dx,dt,xmin,ymin,xmax, ymax::a,
domeinFunctie:: ((a,a) -> (a,a) -> (a,a)-> Bool),aantalIteraties::Int,
werkMatrix:: b (i,i) a}

{- geef de grootste x en y index van het gegeven diffusieprobleem
-}
xbound p = round (((xmax p-xmin p) / dx p) )
ybound p = round (((ymax p-ymin p) / dx p) )

-- geeft de grootste index in de x-richting van de gegeven array
xabound x = b
  where
    ((_,_),(b,_)) = bounds x
-- geeft de grootste index in de y richting van de gegeven array
yabound x = b
  where
    ((_,_),(_,b)) = bounds x

{-
Geeft de werkelijke positie van de gegeven indices weer op het domein
-}
naarPositie p (i,j) = (dx p* (fromIntegral i)+xmin p, dx p* (fromIntegral j)+ymin p)
naarxPositie probleem x = res
  where
    (res,_) = naarPositie probleem (x,0)
naaryPositie probleem y = res
  where
    (_,res) = naarPositie probleem (0,y)

{-
Geeft terug of de gegeven positie tot het domein van het opgegeven
diffusieprobleem behoort
-}
hoortTotDomein p (i,j) | ((i > 0) && (i < xb) && (j > 0) && (j < yb)) =
    domeinFunctie p (naarPositie p (i,j)) (xmin p,ymin p) (xmax p,ymax p)
                        | otherwise = False
  where xb = xbound p
        yb = ybound p

```

```
{-
Zet een iteratiestap aan de hand van een diffusieprobleem en een matrix
-}
zetIteratieStap p =
  p {werkMatrix=losop2d t, aantalIteraties = (aantalIteraties p)+1}
    where
      t = werkMatrix p
      c = (kappa p*dt p)/(dx p*dx p)
      xb = xbound p
      yb = ybound p
      losop2d t
        = array ((0,0),(xb ,yb)) [((i,j),bereken i j)|i<-[0..(xb)],j<-[0..(yb)]]
       where
         o = losop2d t
         bereken i j | hoortTotDomein p (i,j) = res
                     | otherwise = t!(i,j)
                          where res = t!(i,j) + c*(o!(i-1,j) + t!(i+1,j)
                                      + o!(i,j-1) +t!(i,j+1) - 4*t!(i,j))



{-
Vul het domein en de randen met de gegeven vulfunctie en randfunctie voor het
gegeven probleem en geef de array terug met deze waarden
-}
geefMatrix probleem vulFunctie randFunctie
 = array ((0,0), (xb,yb)) [((i,j),vul (i,j))| i <- [0..xb], j <- [0..yb]]
  where
    vul (i,j) | hoortTotDomein probleem (i,j) = vulFunctie positie
              | otherwise = randFunctie positie
      where positie = naarPositie probleem (i,j)
    xb = xbound probleem
    yb = ybound probleem

{-
Domeinfunctie voor een cirkelvormig domein
-}
domeinCirkel (x,y) (xmin, ymin) (xmax,ymax)
              | sqrt(x**2.0 + y**2.0) < (xmax/10)  = False
              | sqrt (x**2.0 + y**2.0) < xmax = True
              | otherwise = False

{-
Zet de waarden van een matrix geassocieerd aan een gegeven diffusieprobleem om
in een string die in een bestand geplaatst kan worden zodat gnuplot met de
waarden om kan gaan.
-}
toonArray probleem = foldr (++) ""
           (concat [[show (naarxPositie probleem i)++" "
                     ++ show (naaryPositie probleem j) ++" "
                     ++show (t!(i,j))++"\n"
                     | j <-[0..(yabound t)]]++["\n"]|i<-[0..(xabound t)]])
   where
     t = werkMatrix probleem

{-
Plaats de waarden van een gegeven matrix geassocieerd aan een gegeven
```

```
    diffusieprobleem in het bestand met de gegeven bestandsnaam
    -}
125 toon bestandsnaam probleem = do file <- openFile bestandsnaam WriteMode
                                   hPutStr file (toonArray probleem)
                                   hClose file

    {-
130 Laat het diffusieprobleem lopen tot het convergeert met de gegeven minimale
    verandering per iteratiestap
    -}
    itereerTotEinde verandering probleem = itereer f (f probleem) probleem
      where
135     f = zetIteratieStap
        grootsteverschil eerste tweede
          = foldl1 max [abs (x!(i,j) - y!(i,j))|i <- [0..(xabound x)],j<-[0..(yabound x)]]
          where
            x = werkMatrix eerste
140         y = werkMatrix tweede
        itereer f x vorige | (grootsteverschil x vorige) < verandering = x
                           | otherwise = itereer f (f x) x

    itereerTijd seconden p = apply zetIteratieStap p (round (seconden/dt p))
145   where
        apply f x n | (0 == n) = x
                    | otherwise = apply f (f x) (n-1)

    geefTijd probleem = dt probleem * fromIntegral (aantalIteraties probleem)
150
    probleem2 vulf randf = probleempje
      where
        start =((geefMatrix  probleempje vulf randf)::Array (Int,Int) Double)
        probleempje = Diff2d 1.12 0.1 0.001 (-1) (-1) 1 1 domeinCirkel 0 start
155 {-
    Voer het programma uit aan de hand van de gegeven programma parameters
    -}
    doe skappa sdx sverandering stijda stijdb sxmin sxmax
            = do toon "data-start.dat" begin
160              toon "data-tijda.dat" eerste
                 toon "data-tijdb.dat" tweede
                 toon "data-einde.dat" einde
                 print $ geefTijd einde
          where
165   begin = probleem
      eerste = itereerTijd tijda begin
      tweede = itereerTijd (tijdb-tijda) eerste
      einde = itereerTotEinde verandering tweede
      -- Zet de strings om in bruikbare waarden
170   kappa = fget skappa
      dx = fget sdx
      verandering = fget sverandering
      tijda = fget stijda
      tijdb = fget stijdb
175   xmin = fget sxmin
      xmax = fget sxmax
      -- genereer de startmatrix
      start =((geefMatrix probleem
                          (\(x,y) -> 0)
```

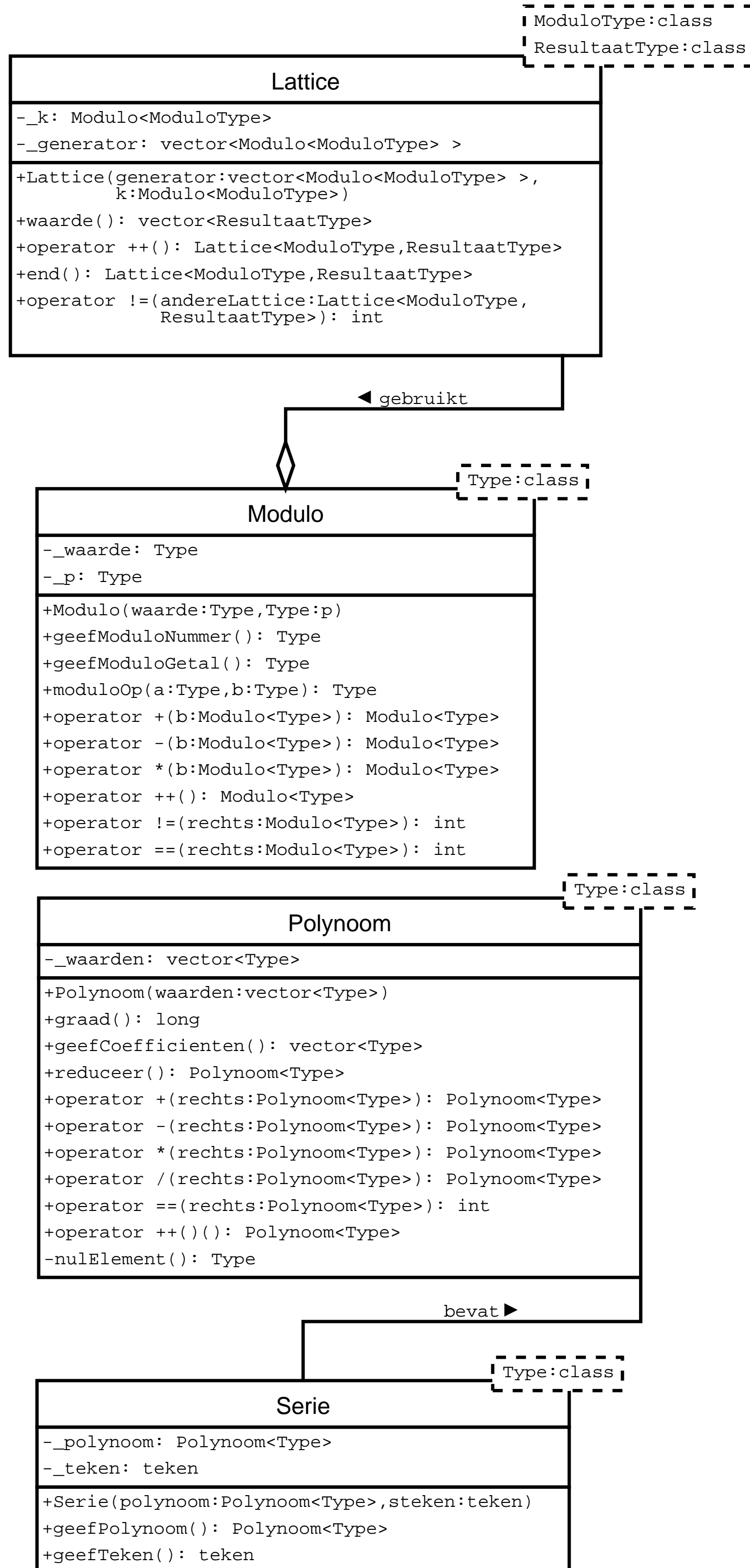
```
180                 (\(x,y) -> if(sqrt(x**2.0 + y**2.0)<0.1)then 20 else 10))
               ::Array (Int,Int) Double)
     -- Definieer het diffusieprobleem
     -- Bemerk dat de definitie van start en probleem aan mekaar gelinkt zijn.
     -- Dit vormt geen probleem
185  probleem = Diff2d kappa dx 0.001 xmin xmin xmax xmax domeinCirkel 0 start
     -- haal een integral argument op op de gegeven x positie in de
     -- argumenten van het programma
     get a = b
       where (b,_) = head (readSigned readDec a)
190  -- haal een floating point argument op op de gegeven x positie in de
     -- argumenten van het programma
     fget a = b
       where (b,_) = head (readSigned readFloat a)

195 main =
     do
     kappaStr <- getLine
     dxStr <- getLine
     veranderingStr <- getLine
200  tijdaStr <- getLine
     tijdbStr <- getLine
     xminStr <- getLine
     xmaxStr <- getLine
     doe kappaStr dxStr veranderingStr tijdaStr tijdbStr xminStr xmaxStr
205 -- toon "data-iep.dat" $ itereerTotEinde 0.5 (probleem2 (\(x,y) -> 0) (\(x,y) -> if(sqrt(x*
```

A.2 Lattice rules

A.2.1 C++ versie

```
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
5
template <class Type> class Modulo;
template <class Type> class Polynoom;
template <class Type> class Serie;
enum teken {POSITIEF,NEGATIEF};
10
/**
 * Toont een modulogetal
 */
template <class Type> ostream& operator<<(ostream &s, const Modulo<Type> &num)
15 {
     return s << "M" << num.geefModuloNummer();
}

/**
20  * Print een vector met getallen in
 */
template <class Type> ostream& operator<<(ostream &s, const vector<Type> &vec)
{
     s << "[";
25   copy(vec.begin(),--vec.end(), ostream_iterator<Type>(s,","));
     s << *(--vec.end()) << "]";
```

Figuur A.2: Klassendiagram van lattice rules



Tabel A.3: Overzicht van de methodes in de C++ klassen voor lattice rules.

(a) Klasse **Lattice**

Methodenaam	
waarde	Geeft het huidige punt in de lattice.
operator ++()	Verandert de **Lattice** zodat **waarde** het volgende punt in de lattice geeft.
end	Geeft een lattice object terug dat aangeeft of het einde van de lattice punten bereikt is.
operator !=	Vergelijkt twee **Lattice**-objecten geeft terug of ze verschillen of niet.

(b) Klasse **Modulo**

Methodenaam	
Modulo	Creëer een **Modulo**-object met opgegeven modulonummer en modulogetal.
geefModuloNummer	Geeft het modulonummer.
geefModuloGetal	Geeft het modulogetal.
moduloOp	Voert de modulo operatie uit.
operator +	Telt twee **Modulo**-objecten op.
operator --	Trekt één **Modulo**-object van een ander af.
operator *	Vermenigvuldigt twee **Modulo**-objecten.
operator ++()	Maakt van een **Modulo**-object zijn opvolger en geef die ook terug.
operator !=	Vergelijkt twee **Modulo**-objecten en geeft terug of ze verschillen van elkaar of niet.

(c) Klasse **Polynoom**

Methodenaam	
Polynoom	Creëer een **Polynoom**-object met opgegeven indices.
graad	Geeft de graad van de polynoom.
geefCoefficienten	Geeft de vector terug van de coëfficiënten van de polynoom.
reduceer	Verwijdert leidende nul-coëfficiënten uit een polynoom.
operator +	Telt twee polynomen bij elkaar op.
operator --	Trekt één polynoom van een ander af.
operator *	Vermenigvuldigt twee polynomen met elkaar.
operator /	Deelt een polynoom door een andere.
operator ==	Vergelijkt twee polynomen en geeft terug of ze gelijk zijn aan elkaar.
operator ++()	Maakt van deze polynoom zijn opvolger
nulElement	Geeft een nulelement van het type waarover de polynoom gedefinieerd is.

(d) Klasse **Serie**

Methodenaam	
Polynoom	Creëer een **Serie**-object met opgegeven polynoom en teken.
geefPolynoom	Geeft de polynoom terug van deze **Serie**.
geefTeken	Geeft het teken van deze **Serie**.

```
        return s;
    }

30  /**
     * Klasse van de getallen van het template type Type modulo een getal van
     * hetzelfde type.
     *
     * Het type wordt doorgegeven met een templateparameter
35   */
    template <class Type> class Modulo
    {
        Type _waarde, _p;
        public:
40      /* Initialiseer modulonummer met de opgegeven 'waarde' en als modulogetal 'p'
         * De opgegeven waarde wordt gereduceerd met de moduloOp methode.
         */
        Modulo(const Type &waarde, const Type &p)
            : _waarde(moduloOp(waarde,p)), _p(p) {}
45
        /* Geef het modulonummer terug
         */
        Type geefModuloNummer() const
        {
50          return _waarde;
        }
        /* Geef het modulogetal terug
         */
        Type geefModuloGetal() const
55      {
            return _p;
        }

        /* Geef een standaardversie van de modulo operatie. Als deze versie niet
60       * gebruikt moet worden dient een nieuwe gedefinieerd te worden. Bv.
         * voor doubles gaat dit als
         * Modulo<double>::moduloOp(double a, double b)
         */
        Type moduloOp(const Type &a, const Type &b) const
65      {
            return a - ((a/b)*b);
        }

        /**
70       * Definieer de standaardoperaties op modulonummers.
         *
         */
        Modulo<Type> operator +(const Modulo<Type> &b) const
        {
75          if (_p == b.geefModuloGetal())
                return Modulo<Type>
                    (moduloOp(this->geefModuloNummer()+b.geefModuloNummer(), _p),
                     _p);
            else
80          {
                cerr << "Bewerkingen met verschillend modulogetal niet ondersteund\n";
                exit(1);
            }
```

```
    }
85
    Modulo<Type> operator -(const Modulo<Type> &b) const
    {
        if (_p == b.geefModuloGetal())
            return Modulo<Type>
                (moduloOp(this->geefModuloNummer()+_p-b.geefModuloNummer(), _p),
90               _p);
        else
        {
            cerr << "Bewerkingen met verschillend modulogetal niet ondersteund\n";
95          exit(1);
        }
    }

    Modulo<Type>  operator *(const Modulo<Type> &b) const
100 {
        if (_p == b.geefModuloGetal())
            return Modulo<Type>
                (moduloOp(this->geefModuloNummer()*b.geefModuloNummer(), _p),
                 _p);
105     else
        {
            cerr << "Bewerkingen met verschillend modulogetal niet ondersteund\n";
            exit(1);
        }
110 }

    /**
     * De prefix ++-operator geeft de opvolger van dit modulonummer
     */
115 Modulo<Type> & operator ++()
    {
        _waarde = moduloOp(++_waarde, _p);
        return *this;
    }
120
    /**
     * Vergelijkt twee modulogetallen
     */
    int operator !=(const Modulo<Type> &rechts) const
125 {
        if ((_p != rechts._p) || (_waarde != rechts._waarde)) return true;
        else return false;
    }

130 /**
     * Vergelijkt twee modulogetallen
     */
    int operator ==(const Modulo<Type> &rechts) const
    {
135     if((_p == rechts._p) && (_waarde == rechts._waarde)) return true;
        else return false;
    }

};
140
```

```
    /*
     * Scalaire vermenigvuldiging
     *
145  * Vermenigvuldigt alle elementen uit een vector van een bepaald type met een element
     * van hetzelfde type.
     *
     */
    template <class Type> vector<Type> operator *(const vector<Type> &vec, const Type &x)
150 {
        vector<Type> resultaat(vec);
        transform(resultaat.begin(),resultaat.end(),
                resultaat.begin(),bind1st(multiplies<Type>(),x));
        return resultaat;
155 }

    /**
     * Genereert een latticepunt op basis van:
     * - een functie v
160  * - een generator z
     * - startwaarde k
     * De types van de generator, het type waarover het modulonummer
     * k gaat en het type van het resultaat moeten opgegeven worden als een
     * templateparameter
165  */
    template <class ModuloType, class ResultaatType>
    vector<ResultaatType> genereerLatticePunt
    (vector<Modulo<ModuloType> >z ,Modulo<ModuloType> k)
    {
170     return (z*k)/k.geefModuloGetal();
    }

    /**
     * Deze klasse genereert een lattice op basis van een generator z en een
175  * startwaarde k.
     *
     * De type waarover het euclidisch domein gaat en het type van het resultaat
     * moeten opgegeven worden als templateparameter.
     *
180  * De klasse Lattice ondersteunt een minimale interface om als iterator te
     * dienen. Dit betekent dat de prefix-operatie ++ een volgend latticepunt
     * genereert dat met de methode 'waarde' kan opgevraagd worden. De methode end
     * geeft een lattice die vergeleken met de operator != aangeeft of het einde van
     * de te genereren latticepunten bereikt is. Zolang de lattice verschilt met de
185  * lattice gegeven door end is het einde nog niet bereikt.
     */
    template <class ModuloType,class ResultaatType> class Lattice {
        Modulo<ModuloType> _k;
        vector<Modulo<ModuloType> > _generator;
190     public:
            /**
             * Maakt een nieuwe lattice aan met een gegeven generator, k en
             * functie v
             */
195         Lattice(const vector<Modulo<ModuloType> > &generator,
                    const Modulo<ModuloType> &k )
                : _k(k),_generator(generator)
```

```
            {
            }
200
            /**
             * Geeft het huidige lattice punt terug
             */
            vector<ResultaatType> waarde() const
205         {
                return genereerLatticePunt<ModuloType,ResultaatType>(_generator,_k);
            }

            /**
210          * Deze operator verhoogt _k zodat een volgend lattice punt kan
             * gegeven worden met 'waarde'
             */
            Lattice<ModuloType,ResultaatType> & operator ++()
            {
215             ++_k;
                return *this;
            }

            /**
220          * Geef een Lattice object terug dat een iterator-achtige
             * end definieert voor het laatste element van een lattice-rule
             *
             * End garandeert het einde van een iteratie met ++ als
             * opvolgers van _k op _k-_k terugkomt. Anders is een dergelijke
225          * iteratie zonder einde.
             */
            Lattice<ModuloType, ResultaatType> end()
            {
                return Lattice<ModuloType, ResultaatType>(_generator,_k-_k);
230         }

            /**
             * Vergelijkt twee lattices en bekijkt of ze dezelfde waarde _k
             * hebben.
235          * Voor de eenvoud laten we de vergelijking met de generators eruit
             */
            int operator !=(const Lattice<ModuloType,ResultaatType> &andereLattice) const
            {
                if(_k != andereLattice._k) return true;
240             else return false;
            }
    };

    /**
245  * klasse van de polynomen
     */
    template <class Type> class Polynoom
    {
        vector<Type> _waarden;
250
        public:
        /**
         * Construeer een polynoom met de waarden uit de gegeven vector.
         */
```

```
255     explicit Polynoom(const vector<Type> &waarden): _waarden(waarden)
        {
        }

        /**
260      * Geeft de graad van de polynoom
         */
        long graad() const
        {
            if (geefCoefficienten().size() > 1) return geefCoefficienten().size()-1;
265         else return 0;
        }

        /**
         * Geeft de vector met de coefficienten terug
270      */
        const vector<Type> & geefCoefficienten() const
        {
            return _waarden;
        }
275
        /**
         * Verwijdert leidende nulcoefficienten uit een polynoom
         */
        Polynoom<Type> &reduceer()
280     {
            while((_waarden.back() == nulElement())  && (graad() > 0))
                _waarden.pop_back();
            return *this;
        }
285
        /**
         * Optelling van twee polynomen
         */
        Polynoom<Type> operator +(const Polynoom<Type> &rechts) const
290     {
            // Indien een van de twee polynomen geen coefficienten heeft
            // geef de rechts terug
            if(_waarden.size() == 0) return rechts;
            if(rechts.geefCoefficienten().size() == 0) return *this;
295
            // bepaal de graad van de resulterende polynoom
            long graadResultaat = graad();
            if (rechts.graad() > graad())
                graadResultaat = rechts.graad();
300         // kopieer de waarden van de polynomen in vectoren
            vector<Type> waardenRechts(rechts.geefCoefficienten());
            vector<Type> resultaat(_waarden);
            // maak de twee vectoren even groot en vul de nieuwe waarden in
            // met nulwaarden
305         waardenRechts.resize(graadResultaat+1,nulElement());
            resultaat.resize(graadResultaat+1,nulElement());

            // tel de twee vectoren op
            transform
310             (waardenRechts.begin(),waardenRechts.end(),
                 resultaat.begin(),resultaat.begin(),plus<Type>());
```

```
        return Polynoom(resultaat).reduceer();
    }

315 /**
     * Aftrekken van twee polynomen
     */
    Polynoom<Type> operator -(const Polynoom<Type> &rechts) const
    {
320     // zet alle waarden uit de rechterhand op negatief
        vector<Type> rechterhand(rechts.graad()+1,nulElement());
        transform(rechterhand.begin(),rechterhand.end(),
                rechts.geefCoefficienten().begin(),rechterhand.begin(),
                minus<Type>());
325     // tel op bij deze polynoom
        return Polynoom(_waarden) + Polynoom(rechterhand);
    }

    /**
330  * Vermenigvuldiging van twee polynomen
     */
    Polynoom<Type> operator *(const Polynoom<Type> &rechts) const
    {
        vector<Type> waarden2 = rechts.geefCoefficienten();
335     // bepaal de graad van de resulterende polynoom
        long graadResultaat = graad()+rechts.graad();
        // vul de resultaatvector met lege waarden
        vector<Type> resultaat(graadResultaat+1,nulElement());
        for (unsigned long i = 0; i < _waarden.size(); i++)
340         for (unsigned long j = 0; j < waarden2.size(); j++)
                resultaat.at(i+j)
                    = resultaat.at(i+j)+ ( _waarden.at(i)*waarden2.at(j));
        return Polynoom(resultaat).reduceer();

345 }

    /**
     * Deelt twee polynomen door elkaar. Geeft het resultaat van een
     * restdeling. De tweede polynoom moet monisch zijn.
350  */
    Polynoom<Type> operator /(const Polynoom<Type> &rechts) const
    {
        // als de linkerkant een nul is of een graad heeft die kleiner
        // is dan rechts geef een nul terug
355     if((graad() == 0) && (_waarden.front() == nulElement())
                ||(graad() < rechts.graad()))
        {
            vector<Type> niets(1,nulElement());
            return Polynoom(niets);
360     }
        else
        {
            // stel de polynoom op die vermenigvuldigd met 'rechts' de
            // linkerhand een graad kleiner maakt
365         vector<Type> phcmacht(graad()-rechts.graad()+1,nulElement());
            phcmacht.back() = geefCoefficienten().back();

            // vermenigvuldig de rechterhand met phcmacht
```

```
            Polynoom andermacht = Polynoom(phcmacht) * (rechts);
370         // stel restant op
            Polynoom restant = (*this) - andermacht;
            // voer de rest van de deling uit op het restant en tel
            // dit bij het resultaat dat we al hadden
            return Polynoom(phcmacht) + (restant/rechts);
375     }

    }

    /**
380  * Vergelijkt twee polynomen
     */
    int operator ==(const Polynoom<Type> &rechts) const
    {
        // verschillende graad
385     if(graad() != rechts.graad()) return false;
        else
        {
            // indien alle twee leeg dan zijn ze gelijk
            if ((_waarden.size() == 0)
390             && (rechts.geefCoefficienten().size() == 0)) return true;
            else
            {
                // voer vergelijking uit
                if (equal(_waarden.begin(),_waarden.end(),
395                     rechts.geefCoefficienten().begin())) return true;
                else return false;
            }
        }
    }
400
    /**
     * Opvolger van een polynoom
     */
    Polynoom<Type> & operator ++()
405 {
        typename vector<Type>::iterator i;
        i = _waarden.begin();
        int stop = 0;
        // verhoog de opeenvolgende coefficienten zolang de verhoging
410     // een nul veroorzaakt
        while((i != _waarden.end()) && !stop)
        {
            ++(*i);
            if (*i == nulElement()) ++i;
415         else stop = 1;
        }
        // als de laatste coefficient ook in een nul veranderde voeg een
        // een toe aan de coefficienten
        if (stop == 0) _waarden.push_back(++nulElement());
420     return *this;


    }

425 private:
```

```
        /**
         * Geef het nulement van type "Type"
         */
        Type nulElement() const
430     {
            return _waarden.front()-_waarden.front();
        }


435 };

    /**
     * Toont een polynoom
     */
440 template <class Type> ostream& operator<<(ostream &s,const Polynoom<Type> &pol)
    {
        s << Serie<Type>(pol.geefCoefficienten(),POSITIEF);
        return s;
    }
445
    /**
     * klasse van de formale Laurent series
     */
    template <class Type> class Serie
450 {
        Polynoom<Type> _polynoom;
        teken _teken;

        public:
455     explicit Serie(const Polynoom<Type> &polynoom,teken steken)
            : _polynoom(polynoom),_teken(steken) {}

        /**
         * Geeft de vector met de coefficienten terug
460      */
        const Polynoom<Type> & geefPolynoom() const
        {
            return _polynoom;
        }
465
        /**
         * Geeft het teken terug
         */
        const teken geefTeken() const
470     {
            return _teken;
        }

    };
475
    /**
     * Deling resulterende in een expansie van negatieve machten
     */
    template <class Type> Polynoom<Type>
480 deling(const Polynoom<Type> &x, const Polynoom<Type> &y,long lengte)
    {
        Type nul = y.geefCoefficienten().front()-y.geefCoefficienten().front();
```

```
    if((lengte == 0) || (x.graad() == 0) && (x.geefCoefficienten().front() == nul)
            ||(x.graad() > y.graad()))
485  {
        vector<Type> niets(0,nul);
        return Polynoom<Type>(niets);
    }
    else
490  {
        if (x.graad() == y.graad())
        {
            vector<Type> resultaat(1,x.geefCoefficienten().back());
            vector<Type> delingX(1,nul);
495          vector<Type> xn(1,x.geefCoefficienten().back());
            vector<Type> delingXrest
                = (x - (Polynoom<Type>(xn) * y)).geefCoefficienten();
            delingX.insert(delingX.end(),delingXrest.begin(),delingXrest.end());
            vector<Type> resultaatDeling
500              = deling(Polynoom<Type>(delingX),y,lengte-1).geefCoefficienten();
            resultaat.insert(resultaat.end(),
                    resultaatDeling.begin(),
                    resultaatDeling.end());
            return Polynoom<Type>(resultaat);
505
        }
        else
        {
            vector<Type> resultaat(1,nul);
510          vector<Type> xverm(x.geefCoefficienten());
            xverm.insert(xverm.begin(),nul);
            vector<Type> resultaatDeling
                = deling(Polynoom<Type>(xverm),y,lengte-1).geefCoefficienten();
            resultaat.insert(resultaat.end(),
515                      resultaatDeling.begin(),
                    resultaatDeling.end());
            return Polynoom<Type>(resultaat);
        }
    }
520 }

/**
 * Toont een serie
 */
525 template <class Type> ostream& operator<<(ostream &s,const Serie<Type> &ser)
{
    if (ser.geefPolynoom().geefCoefficienten().size() < 1) s << "";
    else
    {
530      int macht = 0;
        typename vector<Type>::const_iterator i;
        i = ser.geefPolynoom().geefCoefficienten().begin();
        // toon het eerste element
        s << (*i);
535      ++i;
        ++macht;
        // toon de resterende elementen
        while (i != ser.geefPolynoom().geefCoefficienten().end())
        {
```

```
540             s << " + "<< (*i) << "x^";
                if(ser.geefTeken() == NEGATIEF) s << "-";
                s << macht ;
                macht++;
                i++;
545         }
        }
        return s;
    }

550 /**
     * Definieer de deling op polynomen.
     */
    vector<Serie<Modulo<int> >  >
    operator /(const vector<Modulo<Polynoom<Modulo<int> > > > &z,
555         const Polynoom<Modulo<int> > &n)
    {
        vector<Serie<Modulo<int> > > resultaat;
        vector<Modulo<Polynoom<Modulo<int> > >  >::const_iterator it;
        for (it = z.begin(); it != z.end(); ++it)
560         resultaat.push_back(Serie<Modulo<int> >
                    (deling((*it).geefModuloNummer(),n,101),NEGATIEF));
        return resultaat;
    }

565 /**
     * Definieer de deling op elementen van Z_2
     */
    vector<double> operator /(const vector<Modulo<int> > &z, int n)
    {
570         vector<double> resultaat;
            vector<Modulo<int>  >::const_iterator it;
            for (it = z.begin(); it != z.end(); it++)
            {
                    resultaat.push_back((double)(*it).geefModuloNummer()/(double)n);
575         }
            return resultaat;
    }



580
    /**
     * klasse van de natuurlijke getallen modulo 2
     */
    class Z2: public Modulo<int>
585 {
        public:
            Z2(int waarde): Modulo<int>(waarde,2)
            {}
    };
590 int main ()
    {
        typedef Modulo<int> MN;
        Modulo<int> startWaarde(1,100);

595     vector<MN > vec(3,Modulo<int>(0,100));
        vec.at(0) = Modulo<int>(3,100);
```

```
        vec.at(1) = Modulo<int>(2,100);
        vec.at(2) = Modulo<int>(1,100);
        Lattice<int,double> lattice(vec,startWaarde);
600     do
        {
            cout << lattice.waarde() << ",";
        } while (++lattice != lattice.end());
        cout << "\b\n";
605

        /**
         * Demonstreer polynomial lattices
         */
610     // genereer vectoren om polynomen mee te definieren
        vector<MN> nVec(8,Z2(0));
        nVec.at(0) = Z2(1);
        nVec.at(1) = Z2(1);
        nVec.at(7) = Z2(1);
615     vector<MN> generatorVec(2,Z2(1));
        vector<MN> kVec(1,Z2(1));

        // maak polynomen aan
        Polynoom<MN> nPoly(nVec); // 1+x+x^7
620     Polynoom<MN> generatorPoly(generatorVec); // 1+x
        Polynoom<MN> kPoly(kVec); // 1

        // maak polynomen in een quotientring aan
        Modulo<Polynoom<MN> > generatorPolyMod(generatorPoly,nPoly); // 1+x mod 1+x+x^7
625     Modulo<Polynoom<MN> > kPolyMod(kPoly,nPoly); // 1 mod 1+x+x^7

        // plaats de generator in een vector
        vector<Modulo<Polynoom<MN> > > generator(1,generatorPolyMod);
        // definieer een lattice
630     Lattice<Polynoom<MN>,Serie<MN> > lattice2(generator,kPolyMod);

        // toon de eerste 10 elementen
        int i = 0;
        while (i++ < 10)
635     {
            cout << lattice2.waarde() << ",";
            ++lattice2;
        }
        cout << "\b\n";
640 }
```

A.2.2 Haskell versie

```
    {-# OPTIONS -fglasgow-exts #-}
    {-
    Werkt met hugs mits gebruik van -98 extensions

  5 Uitvoeren onder Hugs:
    runhugs -98 LatticeRules.hs

    Met GHC compileer je met:
    ghc LatticeRules.hs -o naamuitvoerbaarbestand
 10
```

Tabel A.4: Overzicht van de Haskell versie van het latticeruleprobleem

(a) Overzicht van de datatypes

Datatypenaam	
Teken	Teken van een serie.
Polynoom	Een serie met positieve machten. We gebruiken dit datatype voor polynomen. De coëfficiënten worden gescheiden door `:>` en een **Polynoom** moet altijd eindigen op het teken **Z**.
Serie	Een serie met ofwel positieve machten ofwel negatieve machten. Gebruikt **Teken** en **Polynoom**.
Modulo	Een element van een quotiëntring. Het eerste veld is het element, het tweede veld is de n uit $\mathcal{E}/n$.

(b) Overzicht van de klassen en hun instanties

Typeklassenaam	Instanties
Eq => Monoid	**Int**,**Modulo**, **Polynoom**
Monoid => Group	**Int**, **Modulo**, **Polynoom**
Group => Ring	**Int**,**Modulo**, **Polynoom**
Ring => Euclid	**Int**, **Polynoom**
Functor	**Polynoom**
Show	**Polynoom**,**Teken**,**Serie**, **Modulo**
Enum	**Modulo**,**Polynoom**
DivisionType	**Polynoom Serie**, **Int (Ratio Int)**

(c) Overzicht van de functies

Functienaam	
deelPolynoom	Deelt twee polynomen.
reduceer	Verwijdert leidende nulcoëfficiënten uit een gegeven polynoom.
(.*)	Scalaire vermenigvuldiging.
deling	Deelt een polynoom van een kleinere graad door een andere polynoom wat resulteert in een serie met negatieve machten.
generateLattice	Genereert een lattice met de opgegeven generator en startwaarde.
generateLatticePoint	Genereert een latticepunt met de opgegeven generator en waarde.

```
-}

import List
15 import Ratio
import Debug.Trace


{-
20 Aangepaste klassen van Fokker
-}

class (Eq a) => Monoid a where
  (<+>) :: a -> a -> a
25  iszero :: a -> Bool

class Monoid a => Group a where
  neg :: a->a

30 class Group a => Ring a where
  (<*>) :: a->a->a
  isone :: a -> Bool

class Ring a => Euclid a where
35  norm :: a->Int
  divide :: a->a->a
  modulo :: a->a->a
  x `modulo` y | not (iszero y) = x <+> neg((x `divide` y) <*> y)

40 class Ring a => Field a where
  inv :: a-> a


{-
45 Maakt van ingebouwde Haskell integers een euclidische domein
-}
instance Monoid Int where
  (<+>) = (+)
  iszero = (0 ==)
50
instance Group Int where
  neg = negate


55 instance Ring Int where
  (<*>) = (*)
  isone = (1 ==)

instance Euclid Int where
60  norm x = abs x
  divide = div
  modulo = mod


65 {-
Definitie van series en polynomen. Dit is een symplificatie van de
werkelijkheid. Polynomen stellen we voor als een serie met alleen positieve
machten (Polynoom). Een serie kan alleen positieve of negatieve machten hebben en
```

```
    is een Polynoom met een teken.
70  Niet beide. Het eerste getal in een Polynoom is de coefficient voor x^0
    -}
    data Teken = Positief | Negatief

    {-
75  Instantiedeclaratie om een teken (Teken) te tonen
    -}
    instance Show Teken where
      show Positief = ""
      show Negatief = "-"
80
    infixr 5 :>
    data Polynoom a = Z | !a :> Polynoom a

    data Serie a = Serie Teken (Polynoom a)
85

    {-
    Definieer standaardoperaties op Polynomen. Deze zijn equivalent aan de Haskell
    operaties op lijsten.
90  -}
    shead (x0:>xs) = x0
    slast (xl:>Z) = xl
    slast (x0:>xs) = slast xs
    slength Z = 0
95  slength (x0:>xs) = 1+ slength xs
    stake 0 _ = Z
    stake n (x0:>xs) = x0:>stake (n-1) xs


100 {-
    Definieer standaardoperaties op Series. Deze zijn equivalent aan de Haskell
    operaties op lijsten.
    -}
    sstake n (Serie sign x) = Serie sign (stake n x)
105

    {-
    Definieer fmap op een Polynoom
    -}
110 instance Functor Polynoom where
     fmap f Z = Z
     fmap f (x :> q) = f x :> fmap f q

    {-
115 Instantiedeclaratie om Polynomen te tonen
    -}
    instance (Show a) => Show (Polynoom a) where
      show Z = ""
      show x = show (Serie Positief x)
120
    {-
    Instantiedeclaratie voor opvolgers van polynomen of series met positieve machten
    -}
    instance (Group a,Enum a) => Enum (Polynoom a) where
125   succ (x:>xres) | iszero (succ x)  = (x<+> neg x):>(opvxres xres)
```

```
                    | otherwise          = (succ x):>xres
                      where
                        opvxres Z = succ (x <+> neg x):>Z
                        opvxres xres = succ xres
130
{-
Instantiedeclaratie om een Serie te tonen
-}
instance (Show a) => Show (Serie a) where
135  show (Serie _ Z) = ""
  show (Serie sign (x0:>xs)) = show x0 ++ toon 1 xs
    where
      toon _ Z = ""
      toon macht (x1:>xs) = " + " ++show x1 ++ "x^" ++ show sign ++ show macht
140                            ++ toon (macht +1) xs

{-
Instantiedeclaraties voor veeltermen
-}
145 instance (Eq a) => Eq (Polynoom a) where
  Z == Z = True
  (x0:>xs) == (y0:>ys) = (x0 == y0) && (xs == ys)
  _ == _ = False

150 instance (Monoid a) => Monoid (Polynoom a) where
  Z <+> x = x
  x <+> Z = x
  -- reduceer zorgt ervoor dat er geen nullen overblijven op het eind
  (x0:>xq) <+> (y0:>yq) = reduceer ((x0<+>y0):>(xq<+>yq))
155  iszero Z = True
  iszero _ = False

instance (Group a) => Group (Polynoom a) where
  neg Z = Z
160  neg (x:>xs) = (neg x):>neg xs

instance (Ring a) => Ring (Polynoom a) where
  Z <*> x = Z
  x <*> Z = Z
165  (u0:>uq)<*>(v@(v0:>vq))=reduceer $ (u0<*>v0) :> (((u0:>Z)<*>vq) <+> (uq<*>v))

instance (Ring a) => Euclid (Polynoom a) where
  norm x = (slength x)-1
  divide = deelPolynoom
170
{-
Deling van twee polynomen
-}
deelPolynoom Z _ = Z
175 deelPolynoom p q | norm p < norm q = Z
                 | otherwise = phcmacht <+> deelPolynoom (restant) q
                   where
              -- phcmacht is de polynoom die met de rechterhand
              -- vermenigvuldigd en afgetrokken van de linkerhand
180           -- de graad van de linkerhand verlaagt
                     phcmacht = macht verschilmacht phcoef
                     phcoef = (slast p):>Z
```

```
                        verschilmacht = ((slength p) - (slength q))
                        -- vermenigvuldig p met de gegeven macht (m) van x
185                     macht 0 p = p
                        macht m p@(xs:>_) = (xs<+> neg xs):>macht (m-1) p
                   restant = p <+> neg (phcmacht <*> q)

    {-
190 Reduceer reduceert mogelijke nulcoefficienten weg van de hoogste machten
    -}
    reduceer Z = Z
    reduceer (x0:>Z) | iszero x0 = Z
                     | otherwise = (x0:>Z)
195 reduceer (x0:>xres)| (reduceer xres) == Z = reduceer (x0:>Z)
                       | otherwise = x0:> (reduceer xres)

    {-
    Datatype voor een element van een quotientring
200
    Mod x y ~= x modulo y
    -}
    data Modulo a = Mod a a deriving Eq

205 {-
    Instantiedeclaratie voor opvolgers van elementen van een quotientring
    -}
    instance (Enum a, Euclid a) => Enum (Modulo a) where
      succ (Mod x y) = Mod (succ x `modulo` y) y
210
    {-
    Instantiedeclaratie om een modulogetal te tonen
    -}
    instance (Show a, Euclid a) => Show (Modulo a) where
215   -- de modulo operatie kun je niet doorvoeren bij de "constructie" van een
      -- modulotype maar wel bij het tonen
      show (Mod x y) = "M" ++ show (x `modulo` y)

    {-
220 Instantiedeclaraties om van elementen van een quotientring een ring te maken.
    -}
    instance (Euclid a) => Monoid (Modulo a) where
      Mod x xm <+> Mod y ym | ym == xm = Mod ((x<+>y)`modulo` xm) xm
      iszero (Mod x xm) = iszero (x `modulo` xm)
225
    instance (Euclid a) => Group (Modulo a) where
      neg (Mod x xm) = Mod (xm <+> neg (x)) xm

    instance (Euclid a) => Ring (Modulo a) where
230   Mod x xm <*> Mod y ym = Mod ((x<*>y)`modulo` xm) xm
      isone (Mod x xm) = isone (x `modulo` xm)


    {- Scalaire vermenigvuldiging
235 -}
    x .* y = fmap (x <*>) y


    {-
```

```
Typeklasse om het resultaat van de deling vast te leggen
-}
class DivisionType a b where
  (</>) :: [Modulo a] -> a -> [b]

{-
Leg het resultaat bij deling van een polynoom door een polynoom vast
-}
instance (Ring a) => DivisionType (Polynoom a) (Serie a) where
  [] </> y = []
  ((Mod x _):xs) </> y = Serie Negatief (deling x y):(xs</> y)

{-
Voert een deling uit resulterende in een expansie van negatieve machten.
-}
deling Z _ = Z
deling x y | (slength x) == (slength y) =
               xopy :> deling (nul:>(x<+> neg ((xopy:>Z)<*> y))) y
           | otherwise = nul :> deling xverm y
    where
      nul = (shead x) <+> neg (shead x)
      xn = slast x
      xopy = xn
      xverm = nul :> x

{-
Leg het resultaat bij een deling van een geheel getal door een geheel getal vast
-}
instance DivisionType Int (Ratio Int) where
  [] </> n = []
  ((Mod x getal):xs) </> n =  x % n : xs </> n


{-
Genereert een lattice aan de hand van een generator 'z' en een startwaarde 'k'

generateLattice stopt op het moment dat de opvolger van k nul is. In het
polynomiale geval zal generatelattice niet stoppen omdat de opvolger nooit
gelijk aan de nulpolynoom is.
-}
generateLattice z k = punt : volgende
 where
   punt = generateLatticePoint z k
   volgende | not (iszero(succ k)) = generateLattice z (succ k)
            | otherwise = []

{-
Genereert een lattice punt
-}
generateLatticePoint z k@(Mod _ n) = (k.*z)</>n

{-
Handige hulpfuncties om eenvoudig elementen van bepaalde verzamelingen te genereren
-}

-- Maakt van een lijst x een polynoom modulo 1+x+x^7 over Z_2
pol x = Mod (modu x) (modu [1::Int,1,0,0,0,0,0,1])
```

```
    -- maak van een lijst een polynoom over Z_2
    modu [] = Z
300 modu (x:xs) = Mod x 2:>modu xs

    -- maak van een lijst een polynoom over Z_p
    modx [] _ = Z
    modx (x:xs) p = Mod x p :>modx xs p
305
    -- maak van een lijst een vector over E/modv waarbij E de verzameling van x is
    modList [] _ = []
    modList (x:xs) modv = Mod x modv : modList xs modv

310 -- past 'succ' x aantal keer toe
    succs  x 0 = x
    succs  x n = succs  (succ x)(n-1)

    {-
315 genereert de vectoriele lattice over Z_100^3 met generator [1,2,3] en
    startwaarde 1. 'modList' genereert een vector met
    integers modulo een opgegeven waarde (in dit geval 100).

    met
320 fmap (fmap (\x -> (fromIntegral (numerator x)) / (fromIntegral (denominator x)) ))
    kun je het quotient laten uitvoeren
    -}
    main = do print
              (generateLattice (modList [3::Int,2,1] 100) (Mod 1 100)::[[Ratio Int]])
325 {-
    genereert de eerste 10 machtreeksen van de polynomial lattice met generator 1+x en
    startwaarde 1 uit de QuotientRing Z_2[x]/(1+x+x^7)

    'pol' genereert een polynoom in Z_2[x] modulo (1+x+x^7) aan de hand van een
330 gegeven lijst. Met fmap (fmap (sstake 10)) haal je alleen de eerste 10 elementen
    op van elke oneindige machtreeks.
    -}
              print
          (fmap (fmap (sstake 10)) (take 10 (generateLattice [pol [1,1]] (pol[1])))
335                                    ::[[Serie (Modulo Int)]])
```

Bibliografie

E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The fortress language specification, 2005. URL http://research.sun.com/projects/plrg/fortress0785.pdf.

J. J. Barton and L. R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994. ISBN 0201533936.

N. H. F. Beebe, 2004. URL http://www.math.utah.edu/~beebe/software/ieee/.

R. F. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and numerical computing. *IEEE MultiMedia*, 3(2):18–24, 1996. ISSN 1070-986X. doi: http://dx.doi.org/10.1109/93.502291.

Boost.org. Boost C++ libraries, 2005. URL http://www.boost.org/.

D. Bulka and D. Mayhew. *Efficient C++ Performance Programming Techniques*. Addison Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-37950-3.

C++ Standard Committee. Proposed draft technical report on C++ library extensions, 2005. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1745.pdf.

H. Casanova, J. Dongarra, and D. M. Doolin. Java access to numerical libraries. *Concurrency: practice and experience*, 9(11):1279–1291, 1997. URL http://citeseer.ist.psu.edu/casanova97java.html.

B. Dawson, 1997. URL http://www.cygnus-software.com/papers/x86andinfinity.html.

J. W. Demmel and X. Li. Faster numerical algorithms via exception handling. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 234–241, Windsor, Canada, 1993. IEEE Computer Society Press, Los Alamitos, CA. URL http://citeseer.ist.psu.edu/demmel94faster.html.

J. Fokker. Explaining algebraic theory with functional programs. In P. H. Hartel and M. J. Plasmeijer, editors, *Functional programming languages in education (FPLE), LNCS 1022*, pages 139–158, Nijmegen, The Netherlands, Dec 1995. Springer-Verlag.

GHC. The glasgow haskell compiler, 2005. URL http://www.haskell.org/ghc/.

GNU. Gnu C library, 2005. URL http://www.gnu.org/software/libc/manual/html_node/.

D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991. URL `http://citeseer.ist.psu.edu/goldberg91what.html`.

L. Goldthwaite. Technical report on C++ performance, 2003. URL `http://www.open-std.org/jtc1/sc22/wg21/docs/PDTR18015.pdf`.

J. Grattage and T. Altenkirch. A compiler for a functional quantum programming language. Jan. 2005.

N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002. ISBN 0-89871-521-0.

P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/72551.72554.

Hukseflux. Thermal conductivity and thermal properties theoretical background, 2005. URL `http://www.hukseflux.com/thermal%20conductivity/thermal.htm`.

IEEE. Ieee standard for binary floating-point arithmetic, 1985.

IEEE. The single unix specification version 3, 2004. URL `http://www.unix.org/single_unix_specification/`.

ISO. Programming languages - C, 1999. URL `http://www.open-std.org/jtc1/sc22/open/n2794/n2794.pdf`.

JavaGrande. Java grande forum home page, 2003. URL `http://www.javagrande.org/`.

Javagrande, 1998. URL `http://math.nist.gov/javanumerics/reports/issues.html`.

S. P. Jones. Haskell 98 language and libraries: The revised report, 12 2002. URL `http://www.haskell.org/onlinereport/`.

I. JTC1/SC22. Fortran 2003 standard, Nov. 2004. URL `ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1601.pdf.gz`.

W. Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic., Oct. 1997. URL `http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF`.

W. Kahan. William kahan, 2005. URL `http://www.cs.berkeley.edu/~wkahan/`.

W. Kahan and D. Zuras. An open question to developers of numerical software. *Computer*, 38(5):91–94, 2005. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.2005.148.

W. M. Kahan, T. Sumner, D. Gay, and B. A. Wichmann. paranoia, 1998. URL `http://netlib.org/paranoia/index.html`.

J. Karczmarczuk. Structure and interpretation of quantum mechanics: a functional framework. In *Haskell '03: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 50–61, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-758-3. doi: http://doi.acm.org/10.1145/871895.871901.

J. Karczmarczuk. Generating power of lazy semantics. *Theor. Comput. Sci.*, 187(1-2):203–219, 1997. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(97)00065-0.

J. Karczmarczuk. Functional programming and mathematical objects. In *FPLE '95: Proceedings of the First International Symposium on Functional Programming Languages in Education*, pages 121–137. Springer-Verlag, 1995. ISBN 3-540-60675-0.

T. Knapen. Glas: Generic linear algebra software, 2005. URL `http://glas.sourceforge.net/`.

S. B. Lippman. *Inside the C++ object model.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, first edition, 1996. ISBN 0-201-83454-5.

M. Metcalf and J. K. Reid. *Fortran 90/95 explained (2nd ed.).* Oxford University Press, Inc., 1999. ISBN 0-19-850558-2.

N. Miloslav. Haskell, 2006. URL `http://www.zvon.org/other/haskell/Outputglobal/index.html`.

J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, 2000. URL `http://citeseer.ist.psu.edu/moreira99from.html`.

Peer-Directed Projects Center. Irc servers, 2006. URL `http://freenode.net/irc_servers.shtml`.

R. Pozo. Template numerical toolkit for linear algebra: High performance programming with C++ and the Standard Template Library. *j-IJSAHPC*, 11(3):251–263, Fall 1997. ISSN 1078-3482.

R. Pozo and R. Boisvert. Java numerics, 2005. URL `http://math.nist.gov/javanumerics/`.

R. Pozo and B. Miller. Java scimark 2, 2004. URL `http://math.nist.gov/scimark2/`.

A. D. Robison. C++ gets faster for scientific computing. *Comput. Phys.*, 10(5):458–462, 1996. ISSN 0894-1866.

A. Sabry. Modeling quantum computing in haskell. In *Haskell '03: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 39–49. ACM Press, 2003. ISBN 1-58113-758-3. doi: http://doi.acm.org/10.1145/871895.871900.

P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 2003.

P. Selinger. A brief survey of quantum programming languages. *Proceedings of the 7th International Symposium on Functional and Logic Programming*, 2004.

P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control, Apr. 2004.

J. Skibinski. Haskell simulator of a quantum computer. URL `http://www.numeric-quest.com/haskell/QuantumComputer.html`. Gearchiveerd terug te vinden op `http://www.archive.org.`, May 2001.

G. Steel. The soul of a new programming language, 2005. URL http://research.sun.com/minds/2005-0302/.

B. Stroustrup. *The C++ Programming Language, Special Edition*. Addison Wesley Longman Publishing Co., Inc., Boston, 2000. ISBN 0-201-70073-5.

S. Thompson. *The Haskell: The Craft of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0201342758.

A. van Tonder. A lambda calculus for quantum computation, July 2003. URL arXiv:quant-ph/0307150.

T. Veldhuizen, J. Cummings, P. Guio, A. Stokes, S. Shende, B. Kehoe, J. W. Eaton, and M. Mitchell. Blitz++ home page, 2005. URL http://www.oonumerics.org/blitz/.

T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. ISSN 1040-6042. URL http://osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html. Reprinted in C++ Gems, ed. Stanley Lippman.

T. L. Veldhuizen. Techniques for scientific C++, Aug. 1999. URL http://citeseer.ist.psu.edu/298050.html.

Wikipedia. Computer, 2006. URL http://en.wikipedia.org/wiki/Computer.