Het functionele paradigma is de toekomst.

Sjors Sparreboom

Hogeschool Rotterdam
0890040@hr.nl

Johan Bos

Hogeschool Rotterdam
0878090@hr.nl

30 december 2017

Samenvatting

Met de komst van het internet, big data en de toenemende capaciteit van computers is er vernieuwde interesse in de functionele programmeertalen. Dit rapport bekijkt samen met de lezer de voor en nadelen bij de stelling "Het functionele paradigma is de toekomst." Dit gebeurt op basis van een literatuuronderzoek naar de relaties en verschillen van de heersende programmeerparadigma's. Het onderzoek beperkt zich tot het imperatieve en functionele paradigma. Om het functionele paradigma en zijn plaats binnen de informatica beter te kunnen begrijpen is er binnen dit onderzoek van een minimale kennisgeving van de heersende programmeerparadigma's uitgegaan. Ook is kennis van programmeren wenselijk, maar kan het onderzoek ook prima zonder deze kennis worden gevolgd.

I. Inleiding

Voor het maken van programma's wordt er gebruik gemaakt van programmeertalen. Door het gebruik van een programmeertaal is het mogelijk voor een programmeur om een computer te voorzien van instructies en zo de gigantische rekenkracht van het apparaat te benutten. Er zijn door de jaren heen veel programmeertalen ontwikkelt. Sommige revolutionair en vernieuwend, andere een uitbreiding op reeds bestaande talen. Veel van deze programmeertalen delen dezelfde kenmerken en kunnen worden geclassificeerd onder een paradigma.

Het is daarom van essentieel belang dat een keuze voor een paradigma afhankelijk is van een correcte probleem- of doelstelling. Zo wordt er vanuit de academische en wetenschappelijke hoek al jaren aangezet tot een transitie naar het functionele paradigma, maar worden functionele programmeertalen, zoals Haskell en Lisp vaak als een onconventionele, onwerkbare vorm van software ontwikkeling gezien. Echter wordt er momenteel met de komst van het internet en big data een nieuwe efficiënte manier van dataverwerking toegepast die veelal gebruik maakt van parallel, gelijktijdig en gedistribueerd programmeren.

Deze oplossingen schikken zich vaak voor een functionele aanpak van het probleem, maar de achterliggende infrastructuur is vaak imperatief en objectgeoriënteerd waardoor een combinatie veelal uitgesloten is, maar wel wenselijk.

II. Middenstuk

In dit middenstuk zal er een korte uitleg worden gegeven wat de imperatieve en functionele programmeerparadigma's inhouden. Opvolgend zal er worden gekeken naar twee kanten

van deze stelling, namelijk de voor en tegenargumentatie.

Om de plaats van het functionele programmeerparadigma beter te kunnen begrijpen, zal er eerst gekeken moeten worden naar de andere dominante paradigma's. De paradigma's die kort besproken zullen worden zijn: het imperatieve en functionele programmeerparadigma.

i. Imperatief programmeren

Imperatief programmeren houdt in dat een programma een structuur beschrijft volgens het Turingmodel. Dit programma wordt door de computer ingelezen in het geheugen om het in kleine delen sequentieel uit te voeren. De tussentijdse berekeningen worden in een imperatieve taal als data opgeslagen in het geheugen. Deze data kan vervolgens worden opgehaald en gemuteerd door elk volgende instructie die toegang heeft tot deze data, totdat er een antwoord wordt gepresenteerd. Imperatieve programmeertalen zijn door deze eigenschap om de huidige staat van een programma aan te passen zeer geschikt voor het programmeren van computer hardware. Dit is de reden dat de imperatieve stijl van programmeren hedendaags de gangbare manier van programmeren is.

ii. Functioneel programmeren

Functioneel programmeren is een paradigma dat niet nieuw is, maar al sinds de jaren 40 bestaat. Het is een paradigma wat is ontstaan uit het lambda calculus[1], wat Turingvolledig is. Kenmerkend voor functioneel programmeren is het programmeren met functies, het vermijden van globale staat, de niet muteerbare data structuren en een executie model waarin de evaluatie volgorde niet uitmaakt.

III. Algemene voordelen functioneel programmeren

Nu er kort uitleg is gegeven over imperatieve en functionele talen kunnen we kijken naar de verschillen tussen beide paradigma's. Het meest voorkomende paradigma is het imperatief programmeren. Het is echter pas sinds kort dat er hernieuwde interesse is ontstaan in het functionele paradigma vanwege de expressiviteit en het vermogen van deze talen om foutloos gelijktijdig programma's uit te voeren. Deze eigenschappen maken functionele talen geschikt voor het verwerken van de grote hoeveelheden data in een gedistribueerde omgeving. Sinds de komst van het internet en big data zijn de traditionele manieren van dataverwerking niet meer toereikend genoeg. De oplossing: het verdelen van de data over verschillende multi-core processoren, zodat de verwerking van deze data sneller verloopt; dit heet in de informatica gedistribueerd programmeren.

Gedistribueerd programmeren is een complex onderwerp in de informatica en niet zonder enige risico's. Doordat er meerdere multicore processoren betrokken zijn bij de executie van het programma, wat tevens gelijktijdig kan gebeuren, is het cruciaal dat de evaluatie volgorde van het programma klopt. Echter is het moeilijk om deze evaluatie volgorde te waarborgen aangezien de aangesproken processoren niet dezelfde throughput hebben. Hierdoor zijn uitspraken over ruimte en tijd complexiteit moeilijk voor een programmeur. In veel imperatieve programmeertalen zijn locks of mutexes genoemd, geïntroduceerd met als doel het correct afhandelen van deze executievolgorde. Echter zijn de geïntroduceerde oplossingen niet zonder bijwerkingen. Bijwerkingen van deze oplossingen kunnen er voor zorgen dat het programma beland in een deadlock of dat het leidt tot onverwachtse resultaten door data races[13]. Deze bijwerkingen kunnen worden verklaard doordat data in imperatieve programmeertalen muteerbaar is en deze talen inherent sequentieel zijn.

Functionele programmeertalen hebben echter geen van de bovengenoemde problemen. Deze talen niet afhankelijk van de evaluatie volgorde en zijn er geen locks of mutexes, wat ze uitsluit voor deadlocks. Dit maakt het mogelijk om verschillende stukken van een expressie

parallel te evalueren, wat een pre is voor gedistribueerd programmeren. Tevens is de data in de functionele programmeertalen niet muteerbaar, wat inhoudt dat er geen bijwerkingen zijn en data races niet tot nauwelijks voor kunnen komen in deze talen.

Functionele programmeertalen zijn ook erg schaalbaar. Door het gebruik van hogere graads functies kunnen functies gedeeltelijk worden toegepast. Dit maakt het mogelijk om nieuwe functies van oude functies af te leiden, wat ervoor zorgt dat de functies onafhankelijk zijn en zeer schaalbaar. Door de schaalbaarheid van deze talen zijn de algoritmes geschreven in een functionele taal vaak korter en leesbaarder dan in een imperatieve programmeertaal. Door vervolgens een collectie van gerelateerde functies te combineren ontstaan er zogeheten modules, welk herbruikbaar en gemakkelijk te beheren zijn. Functionele talen zijn dus geschikt wanneer er een vaste verzameling *dingen* is en naarmate de code evalueert, er voornamelijk nieuwe bewerkingen toe worden gevoegd aan deze bestaande dingen. Wat deze talen zeer geschikt maakt voor het manipuleren van symbolische data in boom formaat. Dit in tegenstelling tot imperatieve talen, welke zich meer schikken naar een vaste set bewerkingen op *dingen* en naarmate de code evalueert, er vooral nieuwe dingen worden toegevoegd.

IV. Algemene nadelen functioneel programmeren

Dat functioneel programmeren tegenwoordig voor steeds meer situaties en problemen een gangbare oplossing is, zagen we in de vorige sectie. Echter zijn er ook situaties en problemen die zich niet schikken voor een functionele oplossing. Embedded systemen is één van deze gebieden waarin functionele programmeertalen niet geschikt zijn voor het schrijven van programma's (ook wel firmware genoemd).

Embedded systemen zijn gemaakt voor het uitvoeren van een specifieke taak en beschikken vaak over weinig intern geheugen. Ook kunnen deze systemen een andere processor hebben die kan verschillen met de processor die normaliter in een algemene (thuis)computer zit. Door het beperkte interne geheugen en de vele verschillende processoren, zijn functionele programmeertalen niet geschikt voor het schrijven van de firmware voor deze systemen. Functionele programmeertalen gebruiken namelijk aanzienlijk meer geheugen dan hun imperatieve tegenhangers. Data is in deze talen niet muteerbaar en voor elke berekening moet er een nieuw stuk geheugen worden aangesproken.

Doordat er weinig geheugen beschikbaar is in deze systemen en er veel verschillende processorarchitecturen bestaan, is het voor veel programmeurs noodzakelijk om direct het geheugen te kunnen controleren. Echter zijn veel functionele programmeertalen afhankelijk van compiler optimalisatie, waar de compiler vaak geleverd wordt met een garbage collector. Het gebruik van een garbage collector is voor embedded systemen een enorme beperking aangezien de programmeur het onderliggende geheugen niet kan controleren. Door deze beperking zijn de meeste functionele programmeertalen aanzienlijk langzamer ten opzichte van imperatieve programmeertalen en kan het verlies in snelheid worden teruggeleid en verklaard door de focus op abstractie van de functionele talen. Door de focus op abstractie duurt een translatie van een programma naar machine code in een functionele programmeertaal door de compiler vaak langer dan dezelfde translatie gedaan in één van de imperatieve talen.

Een andere veel voorkomend probleem is dat veel van de huidige systemen die momenteel in de omloop zijn, geschreven zijn in een imperatieve taal. Veel van deze systemen zijn al jaren in ontwikkeling en een complete herschrijving van zulke systemen is vaak erg tijdrovend en niet winstgevend. Door vervolgens nieuwe functionele code te koppelen met de huidige imperatieve systemen, ontstaan er koppelingsproblemen.

V. Conclusie

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Referenties

[1] H. Barendregt and E. Barendsen, "The expression problem," Bell Laboratories, Lucent Technologies, Tech. Rep., 2000. [Online]. Available: ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf

[2] P. Wadler, "The essence of functional programming," University of Glasgow, Tech. Rep., 1992.

[3] J. Hughes, "Why functional programming matters," Institutionen för Datavetenskap, Tech. Rep., 1989.

[4] T. software BV, "The software quality company," 2017. [Online]. Available: http://www.tiobe.com/tiobe-index

[5] S. Cass, "The 2017 top programming languages," 2017. [Online]. Available: https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

[6] P. Wadler, "Why no one uses functional languages1," Bell Laboratories, Lucent Technologies, Tech. Rep., 1998.

[7] M. Lipvača, *Learn You a Haskell for Great Good!* No starch press, 2011.

[8] T. Petricek and J. Skeet, *Real World Functional Programming*, 20 Baldwin Road PO Box 261 Shelter Island, NY 11964, 2010.

[9] S. Peyton-Jones and T. Harris, "Programming in the age of concurrency: Software transactional memory," 2017. [Online]. Available: https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

[10] S. Peyton-Jones and E. Meijer, "Haskell is useless," 2006. [Online]. Available: https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

[11] S. Peyton-Jones, "The future is parallel, and the future of parallel is declarative," 2011. [Online]. Available: https://www.youtube.com/watch?v=tC94Mkg-oJU

[12] ——, "Harnessing the multicores: Nested data parallelism in haskell," 2016. [Online]. Available: https://www.youtube.com/watch?v=kZkO3k9g1ps&t=154s

[13] B. Lisper, "Functional programming and parallel computing," Mälardalen University, School of Innovation, Design and Engineering, Tech. Rep., 2013.

[14] P. Wadler, "The expression problem," 1998. [Online]. Available: http://www.daimi.au.dk/~madst/tool/papers/expression.txt

[15] K. Svensson and T. Eliasson, "A comparison of functional and object-oriented programming paradigms in javascript," Blekinge Institute of Technology, Tech. Rep., 2017.

[16] N. S. yan Kanigicharla, "Comparative study of the pros and cons of programming languages java, scala, c++, haskell, vb .net, aspectj, perl, ruby, php scheme," Concordia University, Tech. Rep., 2010.

Terminologie

Big Data:

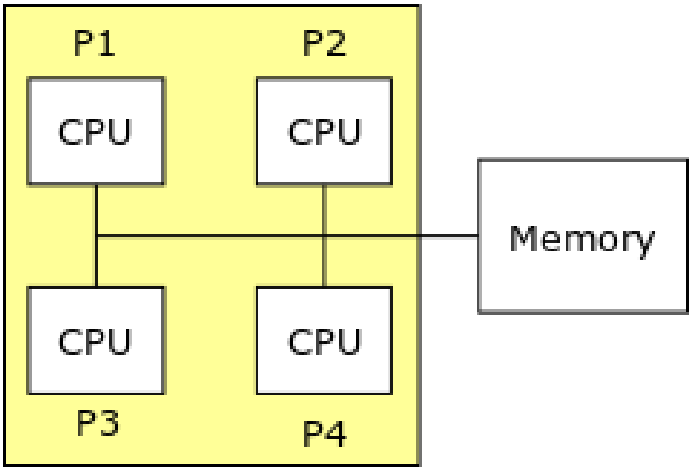
Data sets die zo enorm en complex zijn dat traditionele dataverwerkingssoftware niet langer toereikend zijn.

Lambda calculus:

Het originele model achter de computer, bedacht door de wiskundige Alan Turing in 1936.

Multi-core processoren:

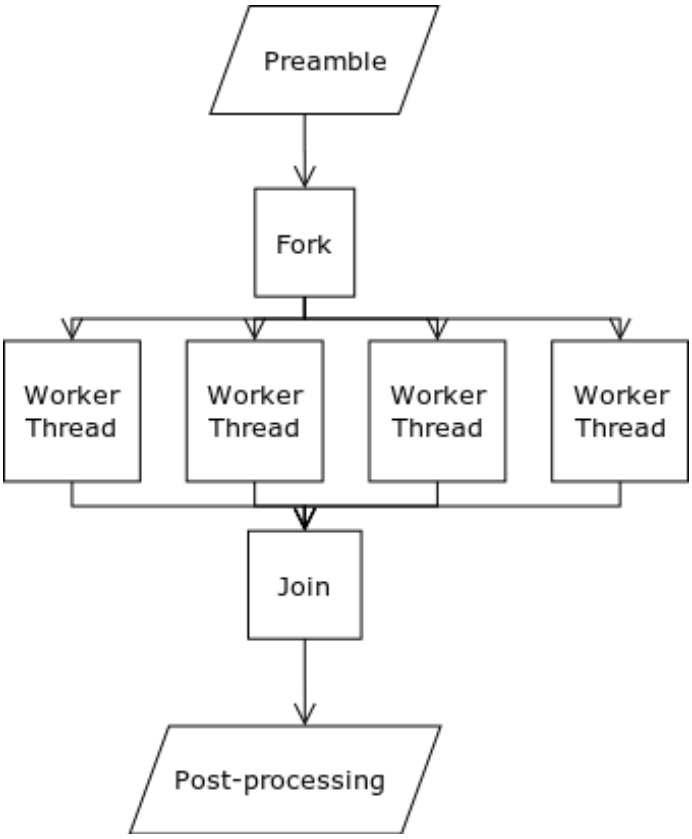
Multi-core processoren, bestaan uit verschillende processoren en gedeeld geheugen, zie figuur 1.



Figuur 1: *Multicore*

Parallel proces:

De verschillende kernen zoals, te zien in figuur 1 kunnen ook parallel verschillende stukken van het programma uitvoeren, zoals te zien in figuur 2. Als verschillende kernen zinnig werk tegelijkertijd doen kan dit het gehele proces versnellen.



Figuur 2: *Parallel proces*

Race conditie:

Twee processen P1 en P2, kunnen beide schrijven naar de variabel tmp. De intentie van de programmeur is dat zowel proces P1 als P2 tmp van een waarde voorziet, deze vervolgens gelijk te gebruiken en een uiteindelijke waarde voor y en z te presenteren. De executie van algoritme 1 levert de volgende waardes voor y en z:

$$y = 24, z = 2055$$

Echter doordat de processen op verschillende snelheden kunnen draaien, doet zich een situatie voor, zoals te zien in algoritme 2 waar de volgorde van executie niet wordt uitgevoerd zoals de programmeur had bedoeld. De executie van dit algoritme levert de volgende waardes voor y en z:

$$y = 4096, z = 2055$$

Zoals te zien is dit niet de juiste waarde voor y. Deze situatie wordt ook wel een race conditie genoemd in de informatica.

Algorithm 1 Programmeurs intentie

Shared Variable
tmp
end Shared Variable

procedure P1	**procedure** P2
.	$tmp = 2048$
.	$z = tmp + 7$
.	.
$tmp = 12$	.
$y = 2 \times tmp$	.
end procedure	**end procedure**

Algorithm 2 Race conditie

Shared Variable
tmp
end Shared Variable

procedure P1	**procedure** P2
$tmp = 12$	.
.	$tmp = 2048$
$y = 2 \times tmp$	.
.	$z = tmp + 7$
.	.
end procedure	**end procedure**

Throughput:

Maximale snelheid waarop *iets* kan worden verwerkt.

Turingmodel:

Het originele model achter de computer, bedacht door de wiskundige Alan Turing in 1936.

Turingvolledig:

Iets is turingvolledig, wanneer het een turing machine kan simuleren.