Het functionele paradigma is de toekomst.

Sjors Sparreboom

Hogeschool Rotterdam
0890040@hr.nl

Johan Bos

Hogeschool Rotterdam
0878090@hr.nl

21 december 2017

Samenvatting

Met de komst van het internet, big data, cloud computing 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 in de industrie. 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 in de industrie 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 big data en cloud computing 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 in de industrie. 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 het programma aan te passen zeer geschikt voor het programmeren van computer hardware. Zo zijn CPU's imperatieve executie machines en compilers vertalen direct naar deze taal. Dit is de reden dat imperatief 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, 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, in het bijzonder de voor en nadelen. Om de verschillen echter goed te begrijpen is het handig om te weten dat op het moment van schrijven van dit rapport het dominante paradigma in de industrie de imperatieve manier van programmeren is. Het is echter pas sinds kort dat er hernieuwde interesse is ontstaan in de functionele talen vanwege de expressiviteit en het vermogen van deze talen om foutloos gelijktijdig programma's uit te voeren. Deze eigenschappen sluiten heel goed aan bij de belangen van veel bedrijven die met de komst van het internet, cloud computing en big data over data beschikken die zo omvangrijk is dat de traditionele data verwerkingssoftware niet toereikend genoeg is om hiermee om te gaan. De oplossing: het verdelen van de data over verschillende multi-core processoren, zodat er betere prestaties kunnen worden behaald en de verwerking 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 aangesproken worden, wat tevens gelijktijdig kan gebeuren, moet er goed nagedacht worden over de evaluatie volgorde. Doordat de aangesproken processoren niet dezelfde throughput hebben zijn uitspraken over ruimte en tijd complexiteit moeilijk voor een programmeur. Zeker als we er vanuit kunnen gaan dat het meest gebruikte paradigma in de industrie de imperatieve stijl van programmeren is. Het imperatief programmeren is inherent sequentieel en data is muteerbaar wat het correct uitvoeren van een programma nog meer bemoeilijkt door eventuele data races. Om data races te voorkomen zijn er in vele imperatieve programmeertalen locks of mutexes geïntroduceerd met als doel het correct afhandelen van de executievolgorde van een imperatief programma. Echter zijn de geïntroduceerde oplossingen niet zonder bijwerkingen, denk

aan de deadlock.

Functionele programmeertalen hebben echter veel van deze problemen niet. Zo zijn functionele 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 kunnen ontstaan, waardoor data races niet tot nauwelijks voor kunnen komen in deze talen.

De functionele programmeertalen zijn ook erg schaalbaar, zo zijn bepaalde algoritmes in functionele programmeertaal vaak minder regels code dan in een imperatieve taal, doordat functionele talen vaak een hogere abstractie hanteren, denk hierbij aan het gebruik van hogere graads functies. Dit zorgt voor een duidelijker en beter herbruikbaar systeem, waar elk stuk code een eigen, gescheiden verantwoordelijkheid heeft, dit heet modulariteit.

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

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

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

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

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

[6] M. LipvaÄ■a, *Learn You a Haskell for Great Good!* No starch press, 2011.

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

[8] 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

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

[10] 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.