Lange tijd behandelde Sun het Java-platform als een ondeelbare eenheid bestaande uit de taal Java, de Java standaard libraries en de Java Virtual Machine (JVM). De afgelopen jaren laten daarentegen een andere beweging zien. Steeds meer alternatieve talen werken met de JVM als doelplatform. Onlangs heeft Sun zelf ook deze stap gezet met JavaFX. Scala bevindt zich in een vergelijkbare positie: een vooruitstrevende taal op een bewezen platform, die zowel objectgeoriënteerde als functionele taalconstructies versmelt in één omgeving.

Scala voor Java ontwikkelaars

Functioneel programmeren buiten universiteit

Scala is een programmeertaal die past in de groeiende rij van talen die op de JVM draaien. De taal is rond 2001 ontwikkeld door dr. Martin Odersky, toen nog hoofdzakelijk als proeftuin voor zijn onderzoek. In 2006 is Scala's implementatie volledig herschreven met als doel de taal productierijp te maken. Sindsdien kan Scala rekenen op een groeiende aanhang. Wat begonnen is als academische proeftuin, is inmiddels uitgegroeid tot een krachtige taal die steeds breder gedragen wordt.

Waarom alternatieve talen?

Voor we Scala gaan verkennen, is het goed om de volgende vraag te stellen: waarom alternatieve talen voor het Java platform? De volgende argumenten worden vaak gebruikt:

1) Meer doen met minder regels code (vooral scripting talen als Groovy en JRuby gebruiken dit argument)
2) De 'multicore challenge': het bieden van een robuust programmeermodel voor parallellisatie (talen met functionele aspecten zoals Clojure zijn hier sterk in)
3) Meer abstractie mogelijkheden en taal features

Op alle bovengenoemde vlakken kan Scala meerwaarde bieden. Ieder argument blijft echter een tweesnijdend zwaard: wat korter, conceptueel beter, of eenvoudiger is, hoeft dat voor de gemiddelde Java-programmeur niet te zijn. Nu duidelijk is dat er bijvoorbeeld geen closures in Java 7 komen, gaat het derde punt voor velen steeds zwaarder wegen. Alternatieve talen bieden het beste van twee werelden: innovativiteit op het taalvlak, gecombineerd met een stabiele, goed presterende JVM aan de onderkant.

Objectgeoriënteerd én functioneel

Scala is, evenals Java, een objectgeoriënteerde en statisch getypeerde taal. In tegenstelling tot Java heeft Scala een uniform objectmodel, dat wil zeggen, er is geen onderscheid tussen primitieve en referentietypes. Verder zijn de mogelijkheden tot compositie van classes uitgebreider dan alleen overerving. De naam Scala, afgeleid van 'Scalable Language', is dan ook gebaseerd op de vele mogelijkheden tot abstractie en modularisatie.

Scala is geen extensie van Java, maar omdat het op hetzelfde fundament bouwt, is de interoperabiliteit met Java naadloos. In codevoorbeeld 1 zien we een Scala class die afleidt van een welbekende Java class, om een servlet te creëren.

```
import javax.servlet.http._

class HelloWorldServlet extends HttpServlet  {
  override def doGet(request: HttpServletRequest,
                     res: HttpServletResponse) =
    response.getWriter().println("Hello World")

}
```

Codevoorbeeld 1: Scala class die een Java class extend.

Het compileren van deze Scala-definitie levert een .class bestand op dat in iedere Java servlet

Sander Mak
studeerde Software Technology aan de Universiteit Utrecht en is momenteel werkzaam als Java-ontwikkelaar bij Info Support BV.

container kan worden gedeployed. Wat direct opvalt, zijn de syntactische verschillen met Java, zoals de underscore voor wildcard imports, het (verplichte) override keyword en het feit dat parametertypes achter identifiers komen in plaats van ervoor. Het def keyword markeert een methodedefinitie, waarvan de implementatie in dit geval bestaat uit één expressie. Meerdere expressies kunnen gegroepeerd worden met curly braces, evenals in Java. Verschil is, dat de laatste expressie in dat block geldt als de returnwaarde; een expliciet return statement is niet nodig.

Naast OO is Scala ook ontworpen vanuit functionele uitgangspunten. Het belangrijkste kenmerk hierbij is dat methodes (functies) ook objecten zijn. Ook de notie van *immutable values* is belangrijk, omdat daarmee het maken van *pure functies* eenvoudiger wordt. Dit zijn functies zonder side-effects, waarbij het resultaat alleen afhangt van de meegegeven parameters en niet van de staat van het programma buiten de functie. Scala geeft de voorkeur aan *immutable* data en collecties, hoewel mutable collecties ook aanwezig zijn.
Hoewel een aantal talen ook OO en functionele paradigma's combineren, zoals OCaml en F# (dat op .Net draait), zijn de OO aspecten in deze talen ondergeschikt aan de functionele. Dit is bij Scala niet het geval.

Scala-essentials

Aan de hand van codevoorbeeld 2 verkennen we de basiselementen van Scala. De class Sorter bevat een methode quicksort die een recursief sorteer algoritme implementeert. In tegenstelling tot Java classes, hebben Scala classes geen expliciete constructor. Wel kunnen properties worden opgegeven, zoals name bij de Sorter class. Properties lijken op class velden, maar krijgen impliciet zogenaamde accessor/mutator methodes, vergelijkbaar met getters en setters die in Java handmatig geïmplementeerd moeten worden. Indien gewenst kunnen ook expliciete implementaties worden gedefinieerd. In de class staat een aanroep naar println. Alle code in een class die niet binnen een methode staat, wordt uitgevoerd tijdens class-instantiatie.

In de quicksort methode vinden we een match/case constructie die later nog besproken wordt. Binnen de tweede case staan twee for loops. Dit zijn echter geen for loops zoals Java die kent, maar *sequence comprehensions*. In het geval van lower lopen we alle elementen van de sublijst tail af (i <- tail), en alleen voor degenen die kleiner zijn dan de integer head (if i < head) wordt het element in de resultaatlijst gezet (yield i). Dit mechanisme werkt niet alleen voor lijsten, maar voor alle types die Iterable implementeren. Ook kan een sequence comprehension over meerdere Iterable's tegelijk werken. Na partitionering van de inputlijst wordt de resultaatlijst in de goede volgorde opgebouwd met behulp van recursieve aanroepen. Als laatste expressie in het block geldt dit als de resultaatwaarde.

```
class Sorter(var name: String) {
  println("Constructing " + name + " sorter.")

  def quicksort(list: List[Int]) : List[Int] = {
    list match {
      case Nil           => Nil;
      case head :: tail => {
          val lower  = for (i <- tail if i <  head)
          yield i;
          val higher = for (i <- tail if i >= head)
          yield i;
                    quicksort(lower)  ++  (head  ::
quicksort(higher));
      }
    }
  }
}

object SorterMain {
  def main(args: Array[String]) = {
    var list   = List(3,1,2)
    val sorter = new Sorter("Main")
    list       = sorter.quicksort(list)
    list.foreach(println)
  }
}
```

Codevoorbeeld 2: Scala object en class in één bestand

De tweede definitie in codevoorbeeld 2 is een objectdefinitie. Hiermee wordt een singleton object gecreëerd. Er bestaat nu één enkele instantie van SorterMain, met een main methode. Evenals Java's main methode is dit het ingangspunt voor een applicatie. De methode is niet statisch zoals in Java. Scala kent namelijk geen statische methodes, aangezien Scala-ontwikkelaars die niet vinden passen in een objectgeoriënteerde taal. Singleton objecten kunnen, indien binnen scope, gerefereerd worden aan de naam. In dit geval zou SorterMain.main(Array()) een correcte aanroep zijn.
In de body van main staan vier statements, waarvan de eerste twee variabelen declareren. Merk op dat twee verschillende keywords worden gebruikt, val en var. Ook hoeft er geen type opgegeven te worden, Scala leidt dit af aan de hand van de toegewezen expressie. Door deze *type-inferentie* wordt dus wel degelijk een type gekoppeld aan de variabele, zonder dat deze hoeft te worden gespecificeerd. Met var wordt een variabele gedeclareerd zoals dat in Java ook gebeurt. Een val declaratie levert echter een *immutable* variabele op. Dit komt overeen met een Java final variabele. In Scala is het goede stijl om waar mogelijk val te gebruiken, omdat dit de hoeveelheid gedeelde muteerbare staat reduceert. Dit voorkomt problemen die gepaard gaan met de anders benodigde synchronisatiemechanismes.

Aan de variabele list wordt een lijst toegewezen met drie integers, ongesorteerd. Vervolgens gebeurt het instantiëren van Sorter net als in Java met het new keyword. Bij het maken van de List ontbreekt het new keyword echter. Deze syntax wordt door de compiler vertaald naar List.apply(3,1,2). Ofwel, List is een object (singleton) met een apply methode en hoeft daarom niet geïnstantieerd te worden. De apply methode accepteert een variabel aantal argumenten, en retourneert deze in een nieuwe instantie van de List class.

Functies als objecten

Met dit laatste mechanisme hebben we direct ook de basis voor functieobjecten gezien in Scala. Een object dat apply implementeert kan met methodeaanroep syntax gebruikt worden. De laatste regel in main van codevoorbeeld 2 laat zien dat ook bestaande methodes meegegeven kunnen worden. In dit geval wordt voor ieder element in list methode println aangeroepen. Het is ook mogelijk om anonieme functies (*closures* of *lambda's* genoemd) te maken. Een gelijkwaardige definitie voor list.foreach(println) is list.foreach(arg => println(arg)), waarbij arg de expliciete parameter van de closure is. De expressie achter de pijl vormt de body van de closure. Onder water vertaalt de compiler het argument voor foreach naar een object met een apply(arg: Int) : Unit = println(arg) methode. Type Unit komt overeen met void in Java. Het feit dat methodes, closures en objecten uiteindelijk allemaal 'first-class citizens' zijn is waar Scala zijn grote uitdrukkingskracht aan ontleent.

Vanuit Java zijn we gewend aan een voorgedefinieerde, niet uitbreidbare verzameling operators. In Scala zijn operators 'syntactische suiker' voor methodeaanroepen. Dat wil zeggen, i < head is speciale infix notatie voor i.<(head). Op deze manier is het dus mogelijk om zelf operators te definiëren, of zelfs bestaande operators te overriden in een subclass. Het zijn dit soort syntactische conventies die het mogelijk maken om natuurlijk aanvoelende libraries te schrijven, of om domeinspecifieke talen in Scala te creëren.

Als laatste illustratie van functieobjecten staat in codevoorbeeld 3 testcode voor quicksort, op basis van het ScalaCheck framework. ScalaCheck maakt het mogelijk om testen te specificeren in termen van eigenschappen van een functie, in plaats van door zelf (unit-)testcode te schrijven. Dit is een ideale testmethode voor pure functies, die verder geen externe afhankelijkheden hebben. Het framework genereert testcases op basis van de specificatie. In codevoorbeeld 3 maken we eerst lokaal een quicksort variabele aan die refereert naar quicksort in Sorter. Deze methode wordt hier niet aangeroepen, omdat de underscore achter quicksort aangeeft dat we een referentie naar de methode willen hebben. Vervolgens worden twee eigenschappen gespecificeerd met ScalaCheck's specifiy methode. Het eerste argument is de naam van de eigenschap, de tweede een closure die de eigenschap karakteriseert.

```
import org.scalacheck.Properties

object SortTest extends Properties("Sorting") {
  val quicksort = new Sorter("Test").quicksort _

       specify("Length",    (list:List[Int])    =>
list.length == quicksort(list).length);
       specify("Sorted",    (list:List[Int])    =>
list.sort(_<_) == quicksort(list));
}
```

Codevoorbeeld 3: Testbare eigenschappen van quicksort.

In de 'Length' eigenschap wordt een functie gespecificeerd die list binnenkrijgt, en vervolgens stelt dat de lengte van list gelijk is aan de lengte van list gesorteerd door quicksort. Een eenvoudige eigenschap van quicksort, maar wel één die geldt voor alle mogelijke lijsten. Het type van list is expliciet opgegeven, zodat ScalaCheck weet wat voor (willekeurige) data het moet genereren om de eigenschap te verifiëren.

De tweede eigenschap is gelijk in structuur, maar stelt dat list gesorteerd met de ingebouwde sort methode van List, gelijk moet zijn aan dezelfde list gesorteerd door quicksort. De standaard sort methode vereist een vergelijkingsfunctie, waarvoor < meegegeven wordt. De underscores vormen daarbij speciale syntax om een operator als functie mee te geven. Als kort intermezzo is het interessant om na te denken over de Java equivalent van list.sort(_<_). Daar zou een anonieme Comparator instantie als parameter geconstrueerd moeten worden. In dit geval bespaart de functionele aanpak de noodzaak voor een specifieke interface, en is de heavyweight syntax van anonieme class instantiatie voorkomen. ScalaCheck genereert nu voor beide eigenschappen 99 testcases met random List[Int] instanties. Wanneer een eigenschap false oplevert, zal dat gemeld worden en kan de bug verholpen worden.

Pattern matching

Codevoorbeeld 2 bevat een match/case constructie die *pattern matcht* op list. Pattern matching is typisch een feature die in veel functionele talen aanwezig is. Het doel van een pattern match is tweeledig: de juiste branch in de code kiezen op basis van de gematchte waarde, en eventuele variabelen (properties) binnen die waarde direct binden. Een van de cases zal uitgevoerd worden. In quicksort bevat match twee alternatieven, Nil (lege lijst) en head :: tail. De laatste introduceert

In Scala zijn operators 'syntactische suiker' voor methode-aanroepen

twee variabele bindingen, die direct in de body van de case gebruikt kunnen worden. Merk op dat :: net als Nil een class naam is, die in dit geval infix gebruikt wordt (::(head, tail) is de alternatieve notatie).
Als we dit vergelijken met Java, zou de implementatie gevormd worden door if-statements met instanceoftests en typecasts, en moeten variabelen expliciet uit de class opgevraagd worden. Als je dit verder doorvoert, kom je op het Visitor pattern uit, wat in Scala door pattern matching overbodig

Pattern matching vormt ook de basis voor Scala's antwoord op de 'multicore challenge', namelijk actor-based concurrency. Een actor is een object met een mailbox, die (immutable) berichten ontvangt van andere actors. Deze berichten worden ontleed met pattern matching, waarna de actor actie onderneemt en eventuele nieuwe actors start. Er is verder geen gedeelde data, alleen berichtenverkeer. Daarom is er geen synchronisatie logica nodig, en schaalt dit model veel makkelijker over meerdere cores dan lock-based concurrency. Dit lijkt erg op hoe Erlang concurrency aanpakt, met dit verschil dat actors in Scala eenvoudigweg een library vormen, zonder speciale taal-ondersteuning.

Pattern matching vormt ook de basis voor Scala's antwoord op de 'multicore challenge'

Mixin compositie

Naast classes kent Scala *traits*. Traits zijn vergelijkbaar met Java interfaces. Echter, traits kunnen in tegenstelling tot Java interfaces wel velden bevatten, en implementaties voor methodes. Een Scala class kan één of meerdere traits extenden. Codevoorbeeld 4 laat zien hoe dat in zijn werk gaat, door een vereenvoudigde versie van een tweetal traits uit Scala's standaard library te presenteren.

```
trait Ordered[A] {
  def compare(that: A): Int

  def < (that: A): Boolean = this.compare(that) <0
  // ... meer operators ...
  def compareTo(that: A): Int = compare(that)
}

trait Iterable[A] { /* next(), hasNext() etc. */ }

class String extends Ordered[String] with
Iterable[Char] {}
```

Codevoorbeeld 4: Traits en mixin compositie.

Het voorbeeld bevat één concrete class (String) die twee traits gebruikt. Dit mechanisme heet mixin compositie en biedt een beperkte vorm van multiple inheritance. Er mogen willekeurig veel with clausules opgegeven worden. De trait Ordered bevat zowel een abstracte methode (compare) als concrete methodes (< en compareTo). Deze concrete methodes mogen geïmplementeerd worden in termen van de abstracte methode compare, omdat een trait nooit alleen geïnstantieerd kan worden. Dit gebeurt altijd door middel van mixin compositie, waarbij de concrete class een implementatie moet geven. In dit geval zal de String class compare moeten implementeren en de methodes uit Iterable. Ook de this referentie binnen Ordered verwijst niet naar een instantie van Ordered zelf, maar naar een instantie van de class die Ordered uiteindelijk gebruikt.

Wat mixin compositie verder onderscheidt van Java's overerving is dat het ook mogelijk is om tijdens het instantiëren van een class traits in te mengen. Een voorbeeld: new String("string") with RichIterator[Char] geeft een String met methodes gedefinieerd in de (fictieve) RichIterator trait. Dit mechanisme is een krachtige vorm van late binding, die onder meer gebruikt kan worden om dependency injection te bewerkstelligen zonder een framework te moeten gebruiken.

Mixin compositie brengt modulariteit op een hoger niveau. Deze modulariteit zie je op veel vlakken terug. Ongeveer alles in Scala kan recursief genest worden (packages in packages, classes in classes enzovoorts). De combinatie van mixins en andere vormen van modulariteit binnen Scala biedt ontwikkelaars zeer veel mogelijkheden om zichzelf niet te hoeven herhalen.

Type systeem

Het doel van typesystemen is om compiletime bepaalde categorieën van problemen te voorkomen. Welke dat zijn, is afhankelijk van de uitdrukkingskracht van het typesysteem. Zo is Java's typesysteem krachtiger geworden na de toevoeging van generics. Scala's typesysteem is nog geavanceerder. Vooral op dit vlak is duidelijk dat de taal een product is van academisch onderzoek. Toch lijkt het typesysteem in eerste instantie veel op dat van Java. Ook Scala kent (sub)classes en generics, maar dan in uitgebreidere vorm. Alle Java types zijn geldige Scala types. Vanuit dit gemeenschappelijke vertrekpunt divergeert Scala's typesysteem.

Functietypes zijn aanwezig om functies te typeren. Zo heeft quicksort het type (List[Int]) => List[Int], en het type van < is (Int,Int) => Boolean. Het moge duidelijk zijn dat functietypes enorm kunnen groeien. Type-inferentie is dan ook essentieel en biedt de voordelen van dynamische talen (minder code) maar wel met de veiligheid van statische typering. Scala biedt echter een beperkte vorm van type-inferentie, namelijk alleen lokaal. Daarom moeten bijvoorbeeld parameter-types wel worden opgegeven (aanroepende context wordt niet geraadpleegd), maar kan het returntype vaak lokaal afgeleid worden.

Verder biedt het typesysteem *tuples*, een mogelijkheid om types te bundelen zonder een expliciet wrappertype te introduceren. Een functie kan hiermee gelijktijdig meerdere resultaten retourneren, door als returntype bijvoorbeeld (Int, String) te definiëren. Als laatste is het mogelijk om *view bounds* aan te geven op types. Een bound geeft aan dat wat voor type er ook binnenkomt, deze converteerbaar moet zijn naar het gewenste view bound type. Vervolgens kunnen conversie functies impliciet worden doorgegeven op basis van scoping. De precieze werking voert te ver om in dit artikel te behandelen, maar het is een belangrijk mechanisme om standaard Java classes te decoreren met extra Scala functionaliteit zonder dat ontwikkelaars daar iets voor hoeven te doen.

Het typesysteem van Scala heeft een hoge complexiteit (veel features zijn nog niet genoemd), maar ontwikkelaars worden daar in principe niet volledig aan blootgesteld. Het biedt vooral library-ontwikkelaars de mogelijkheid om veel constraints in types vast te leggen, waar ontwikkelaars automatisch van profiteren door type-inferentie. Traditioneel gezien staan types centraal in functionele programmeertalen en Scala vormt daar geen uitzondering op.

Aan de slag met Scala

De SDK biedt zowel een compiler als een interactieve shell. In deze shell kunnen snel korte code fragmenten worden uitgeprobeerd, met directe feedback zoals veel scripting talen dat ook kunnen. Gecompileerde Scala programma's kunnen met het scala commando uitgevoerd worden, of met de JVM waarbij de Scala runtime libraries op het classpath moeten staan.

Er zijn plugins beschikbaar voor zowel Eclipse, Netbeans als IntelliJ. Ondersteuning voor Maven is ook beschikbaar in de vorm van een Scala plugin en een Scala archetype. De plugin ondersteunt compilatie van gemengde Java/Scala projecten. Op deze manier is Scala eenvoudig te integreren in bestaande Java projecten, waarbij circulaire referenties tussen Java en Scala classes geen probleem vormen.

Uiteraard kan geen enkele zichzelf respecterende taal zonder een uitgebreid webapplicatie framework. Scala vormt geen uitzondering en biedt het Lift framework, dat optimaal gebruik maakt van alle taalfeatures. Zo gebruikt de geïntegreerde O/R mapper nauwelijks reflectie dankzij Scala's expressieve typesysteem. Lift's implementatie is servlet gebaseerd en kan daarom in bestaande Java applicatieservers gedeployed worden.

Een brandende vraag is natuurlijk of Scala volwassen genoeg is om daadwerkelijk te gaan gebruiken. Odersky laat duidelijk weten dat wat hem betreft Scala definitief uit de academische ivoren toren is gekomen. De evolutie van de taal zal dan ook zorgvuldig gepland worden. Experimentele features worden in een aparte branch gehouden, in tegenstelling tot de beginjaren, waarin de taal bij iedere minor update wel brekende veranderingen bevatte. Bovendien is er een stabiele Scala Language Specification, die in formaliteit de Java Language Specification voorbij streeft. Onlangs is ook het boek 'Programming in Scala' verschenen, dat een uitgebreid overzicht van de taal geeft.

Verschillende bedrijven zetten Scala in op bedrijfskritische toepassingen. Zoals Twitter, dat nieuwe core componenten in Scala ontwikkelt en integreert in het bestaande Java landschap, en SAP, dat het Lift framework gebruikt voor community portals. Kortom, de eerste schreden op het 'enterprise' vlak zijn al gezet.

Conclusie

Een nieuwe taal leren is altijd lastig, zeker wanneer het een taal zo rijk aan features als Scala betreft. Voor Java-ontwikkelaars is het wel een taal die aansluit op een bekende omgeving. Het is mogelijk om Scala te schrijven zoals je Java zou schrijven, daarbij gebruikmakend van bestaande Java-code. Verder is de uitdaging om gaandeweg meer idiomatische Scala-code te schrijven. Dat wil zeggen, gebruik maken van functionele constructies, mixin compositie en andere mogelijkheden. Al met al een graduele transitie, om uiteindelijk alle voordelen van Scala te benutten.

We hebben gezien dat Scala's functionele inslag en expressieve typesysteem leiden tot code die compacter is dan vergelijkbare Java code. Functioneel programmeren in Scala is geen must, maar wordt sterk aangemoedigd vanuit de taal. Immutability gecombineerd met functioneel programmeren vormt een natuurlijke bodem voor robuuste en testbare code, ook in parallelle omgevingen. Veel innovatieve concepten, zoals ScalaCheck en actor-based concurrency, komen voort uit deze mogelijkheden. Dit maakt de community rond Scala zeer divers en vooruitstrevend. Programmeren in Scala is dan ook uitdagend, en zeker de moeite waard. «

Referenties

Scala site: http://www.scala-lang.org
Blogs: http://scala-blogs.org/
Scala wiki: http://scala.sygneca.com
ScalaCheck: http://www.scalacheck.org
Lift web framework: http://liftweb.net