ACHTERGROND

Formele methoden leggen fout in Java's sorteeralgoritme bloot

Frank de Boer is hoofd van de formele-methodengroep van het Centrum Wiskunde & Informatica (CWI). Stijn de Gouw is een postdoc in een privaat-publieke samenwerking tussen CWI en SDL-Fredhopper. Het onderzoek werd uitgevoerd binnen het EU-project Envisage. Een uitgebreide analyse van het werk is te vinden op de homepage van dit project.

1 May 2015

Afgelopen februari dienden onderzoekers van het Centrum Wiskunde & Informatica een bugreport in voor Java. Het standaard sorteeralgoritme, Timsort, bleek in specifieke gevallen te kunnen crashen. De fout kwam aan het licht toen ze het algoritme met formele methoden doorlichtten. Frank de Boer en Stijn de Gouw van het CWI leggen de kwestie uit.

In 2002 ontwikkelde Tim Peters een nieuw hybride sorteeralgoritme voor de Python-programmeertaal. Hij combineerde ideeën uit traditionele *merge sort*– en *insertion sort*-algoritmes en maakte handig gebruik van het idee dat datasets in de dagelijkse praktijk vaak al gedeeltelijk gesorteerd zijn. Bovendien buit het algoritme op slimme wijze de cache- en geheugenarchitectuur van computers uit, waardoor het zeer efficiënt werkt. Vanwege deze aantrekkelijke eigenschappen werd Timsort al snel vertaald naar Java (als

java.util.Collections.sort en *java.util.Arrays.sort*).

Vandaag de dag is Timsort wijdverspreid als het standaard sorteeralgoritme in Java. Het is dus van groot belang dat het correct is en niet crasht of andere problemen veroorzaakt. Omdat het daarnaast vrij complex is, vonden we dit een mooie uitdaging om onze tanden in te zetten. Begin dit jaar hadden we succesvol een project afgerond om *counting*- en *radix sort*-implementaties in Java formeel te analyseren, met ondersteunende software hiervoor, en nu wilden we een volgende stap zetten.

Maar tot onze verrassing bleken we niet in staat de correctheid van de Timsort-implementatie in Java te bewijzen. Een nadere analyse bracht hiervoor een simpele reden naar voren: Timsort bleek een fout te bevatten. Onze bewijspogingen zette ons uiteindelijk op het spoor van de fout, die overigens ook in de originele Python-implementatie voorkomt.

Een taal als Java wordt veel gebruikt. Denk maar aan de honderden miljoenen Android-telefoons die er zijn. Het is dus belangrijk dat er geen crashes optreden.

Timsort herschikt de inputreeks van links naar rechts door te zoeken naar opeenvolgende subreeksen van reeds gesorteerde

elementen – *runs* in Timsort-terminologie – en door deze op het juiste moment samen te voegen. Daarbij kijkt het algoritme enkel naar de lengte van de runs; de concrete inhoud speelt verder geen rol.

Het algoritme zorgt er daarom eerst voor dat de runs een minimum lengte hebben; is een run te kort, dan wordt deze via insertion sort uitgebreid. Vervolgens gaat de methode *mergeCollapse()* aan de slag met het samenvoegen van deze runs. Daar wordt net zo lang mee doorgegaan totdat de lengtes, die apart in een array runLen worden bijgehouden, aan de volgende twee condities voldoen:

1. runLen[n-2] > runLen[n-1] + runLen[n]
2. runLen[n-1] > runLen[n]

De eerste conditie stelt dat de laatste twee runs samen korter moeten zijn dan de twee-na-laatste run (dit volgt het patroon van de bekende Fibonacci-reeks). De tweede stelt simpelweg dat de voorlaatste run langer moet zijn dan de laatste (dus dat de lengtes steeds kleiner moet worden). Zolang een van de condities niet opgaat, voegt mergeCollapse() twee van de laatste drie runs samen.

De veronderstelling van Peters was dat hierdoor een invariant geldt: dat de twee condities gelden voor *alle* runs in de lijst. Deze invariant garandeert daardoor dat de lengtes van alle runs exponentieel afnemen (zelfs sneller dan de Fibonacci-reeks). Zelfs een heel grote invoer kan zodoende worden afgedekt door een zeer beperkt aantal runs, en een beperkte geheugenruimte voor de runLen-array volstaat.

Toen we echter formeel probeerden te bewijzen dat die invariant geldt, bleek dat niet het geval te zijn. Het gevolg is dat er bij bepaalde invoer te veel korte runs worden gegenereerd, meer dan op basis van de invariant kan worden verwacht. Het aantal runs past daardoor niet meer in de beperkte ruimte die gereserveerd is voor runLen, en Timsort crasht met een

ArrayOutOfBoundsException.

Gelukkig kan de fout door een relatief kleine en lokale aanpassing van de code worden hersteld. Als we de voorwaarden in mergeCollapse() testen op de laatste *vier* runs in plaats van de laatste drie, dan kunnen we bewijzen dat de invariant wel geldt. Deze correctie is ondertussen overgenomen in Python.

Een andere mogelijkheid is de maximale lengte van runLen voldoende te vergroten. Hoe veel groter dat moet zijn, hebben we aangetoond door middel van een worstcaseanalyse. De Java- en de Android-versies van Timsort gebruiken deze analyse om de fout te corrigeren.

Duidelijk begrensd

Voor ons bewijs gebruikten we Key, een softwaresysteem ter ondersteuning van de analyse van sequentiële Java- en Javacard-toepassingen dat is ontwikkeld in een samenwerking tussen verschillende Duitse en Zweedse universiteiten. Met behulp van Key kan de correctheid van een programma ten opzichte van een specificatie formeel worden bewezen.

Zo'n specificatie bestaat in het algemeen uit precondities (voorwaarden aan de invoer) en postcondities (voorwaarden aan de uitvoer) voor alle methodes. Voor een sorteeralgoritme kan een preconditie simpelweg stellen dat de invoer een niet-lege reeks moet zijn, en de postconditie dat het resultaat een gesorteerde permutatie van de invoer is.

Key gebruikt als specificatietaal de Java Modeling Language (JML). Deze taal bestaat overwegend uit normale Java-expressies en is daardoor makkelijk te begrijpen voor Java-programmeurs. Er is slechts een handvol uitbreidingen, zoals *kwantoren* (*forall T x*, *exists T x*) die nodig zijn om bijvoorbeeld de invariant uit te drukken die een eigenschap beschrijft van *alle* runs. JML-specificaties worden in het commentaar bij de methodedeclaratie in het .java-bestand gezet.

Voor mergeCollapse kunnen we bijvoorbeeld (versimpeld) de volgende specificatie opstellen:

```
/*@ requires
  @   stackSize > 0;
  @ ensures
  @   (forall int i; 0<=i && i<stackSize-2;
  @         runLen[i] > runLen[i+1] + runLen[i+2])
  @   && runLen[stackSize-2] > runLen[stackSize-1]
  @*/
private void mergeCollapse()
```

De preconditie (de *requires*-clausule) 'stackSize > 0' betekent dat mergeCollapse() alleen kan worden aangeroepen wanneer minstens één run is toegevoegd. De twee formules in de postconditie (de *ensures*-clausule) impliceren dat alle runs voldoen aan de twee condities wanneer mergeCollapse() klaar is.

Verder kunnen er in JML nog klasse-invarianten worden toegevoegd voor het uitdrukken van algemene condities op de waarden van de klassevariabelen. Die hebben doorgaans betrekking op dataconsistentie of randsituaties. Klasse-invarianten gelden vóór en na elke aanroep van een methode.

Met behulp van Key kan formeel worden bewezen dat wanneer de methode wordt aangeroepen met een willekeurige invoer die aan de preconditie voldoet, de methode normaal eindigt en de uitvoer na afloop voldoet aan de postconditie. De tool is dus in staat om de methodespecificaties statisch te bewijzen voor alle mogelijke invoer, zonder dat het programma zelf wordt uitgevoerd.

Key gebruikt hiervoor symbolische uitvoering: de methode wordt met symbolische waarden uitgevoerd, waardoor alle mogelijke executiepaden worden beschouwd. Als de methode voor elke input bewezen normaal eindigt, staat dat bekend als *totale correctheid*. Uiteraard voldoet Openjdk's *java.utils.Array.sort* hier niet aan, want deze methode termineert bij sommige invoer met een exceptie.

Contracten alleen bieden echter geen sluitende oplossing voor formele verificatie. Wanneer de methode namelijk loops bevat die niet vooraf duidelijk begrensd zijn, zal de methode met symbolische uitvoering nooit eindigen. Voor de symbolische uitvoering van loops moet Key daarom over een speciaal soort invariant beschikken die geldt vóór en na elke iteratie en

bovendien de postconditie kan bewijzen. Op het moment dat zo'n invariant is gevonden, hoeft de tool de loop nog maar één keer symbolisch uit te voeren.

Zo'n geschikte loop-invariant moet echter eerst worden gevonden. Dat is en blijft gedeeltelijk mensenwerk, waarbij goede toolondersteuning onontbeerlijk is. De ontwerpers van Timsort dachten te kunnen volstaan zonder dergelijke ondersteuning, maar gingen daarmee dus de mist in.

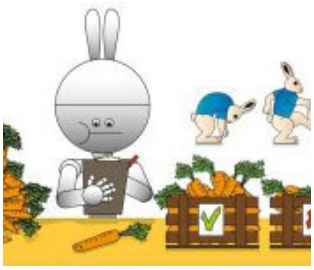
Facebook

Tot nog toe is dit een van de moeilijkste en grootste correctheidsbewijzen van een bestaand Java-programma. Er waren meer dan twee miljoen logische regels en duizenden handmatige stappen nodig. Uitgebreid testen, handmatige codereviews en dergelijke waren niet bij machte deze fout te detecteren. *Model checking*, dat gebaseerd is op een exploratie van alle mogelijke berekeningen, biedt bij dergelijke algoritmes ook geen soelaas. Er moeten namelijk kunstmatige limieten aan bijvoorbeeld het aantal loop-iteraties worden opgelegd om het aantal mogelijke berekeningen eindig te maken.

Onze analyse toont daarom het cruciale belang van formele bewijsmethoden aan. Er zijn nog enkele andere recente voorbeelden te vinden van succesvolle toepassingen van formele methoden. Facebook heeft bijvoorbeeld de Separation Logic Analyzer van Microsoft omarmd om C-code te controleren op geheugenlekken, *dangling pointers* en *double frees*. En techneuten bij Amazon Web Services gebruiken Temporal Logic of Actions (TLA), een techniek ontwikkeld door Leslie Lamport, die vorig jaar de Turing Award ontving.

Edited by Pieter Edelman

Related



BACKGROUND 23 May

Tea and coffee with Rik and Derk-Jan – The impact of AI on testing

Artificial intelligence is a popular topic these days. As companies are including it in their

OPINION 22 May

Better every day

Especially in the embedded systems industry, the approach has traditionally been to build products that get worse over time.

OPINION 16 May

How digitalization disrupts companies

Although everyone talks about digitalization (software, data and AI) and the risk of disruption that this

BACKGROUND 9 May

Using AI to streamline remote communication in healthcare

Philips researchers have already been using artificial intelligence

Bits&Chips strengthens the high tech ecosystem in the Netherlands and Belgium and makes it healthier by supplying independent knowledge and information.

Bits&Chips focuses on news and trends in embedded systems, electronics, mechatronics and semiconductors. Our coverage revolves around the influence of technology.

Contact

Techwatch bv
Novio Tech Campus
Transistorweg 7h
6534 AT Nijmegen
The Netherlands
+31 24 3503532
info@techwatch.nl

Service

- Contact
- Advertising
- Newsletter
- Membership

Our events

- Dutch Machine Learning Conference
- High Tech Systems
- Trends and Challenges in Reliability
- Dutch System Architecting Conference
- Software-Centric Systems Conference
- Dutch Machine Vision Conference
- Benelux RF Conference

Our websites

- High Tech Institute
- Mechatronica&Mach
- Techwatch Books