

Sidoeffektsanalys för referens-attribut-grammatik

Mikael Johansson
Lunds Universitet
2017

Sidoeffekter

Det finns situationer då det är viktigt att veta att metoder inte har några sidoeffekter. Det går då att genomföra olika optimeringar, bland annat för kompilatorer, utan att orsaka fel.

Avsaknad av sidoeffekter kan visa att metoder inte påverkar resultatet av varandra. Anropsordningen kan då optimeras och delresultat kan memoriseras, det vill säga sparas och återanvändas, vilket annars kan leda till fel. En sidoeffektschecker kan användas för att undvika sådana fel.

Jag har programmerat en sidoeffektschecker för Java specifikt för att användas med JastAdd. JastAdd är ett verktyg för att konstruera kompilatorer med hjälp av en formalism kallad referens-attribut-grammatik. JastAdd ställer krav på olika Javametoder som att de måste vara sidoeffektsfria och producera nya objekt. Dessa krav finns eftersom JastAdd använder tekniker som kräver sidoeffektsfrihet. Ingen verifiering görs av dessa krav vilket lämnar risken för att man gör fel och det är det som jag försökt att åtgärda.

Jag testade först om diverse tidigare verktyg för Java kan användas även för JastAdd. Jag bedömde att ingen av de testade alternativen fungerade riktigt för JastAdd. En flesta klarar inte av memorisering eller de andra specialfallen som dyker upp vid användande med JastAdd. De mest kapabla verktygen är för långsamma och svåra att arbeta med.

Jag började därför programmera ett nytt verktyg som är anpassat efter de krav som JastAdd ställer. Jag designed ett nytt typsystem med annoteringer utformade för att hantera de krav JastAdd ställer.

Mitt verktyg genomför en snabb analys som kan hitta många vanliga fel som nybörjare till JastAdd kan göra. Även för professionella projekt kan sidoeffekter som har introducerats möjligen upptäckas.

Mitt bidrag är ett verktyg som kan användas för att upptäcka vanliga sidoeffekts fel in Java och som inte är för svårt att manuellt annotera för. Annoteringarna är designade med tanke på svårighet att använda annoteringarna, analyskostnad och vilka olika beteende man troligen vill tillåta. Jag kombinerade tidigare idéer från andra verktyg som JPure och OpenJML tillsammans med nya idéer för att hantera JastAdd-relaterade situationer.

Typsystemschecker

Jag provade flera olika typsystemsverktyg men de hade alla olika nackdelar. JPure som blev min primära inspiration

använde egna annotering för sidoeffektfrihet så som *@Fresh* för nyskapande objekt, *@Local* för lokala ändringar och så vidare. JPure's annoteringar kändes nära till beteendena som jag behöver kategorisera metoder i.

Example 1. Olika annoterade metoder.

```
@Secret(group=X) X value=null; int counter=0;

@Pure(group=X) public X method(){
    if (value!=null)
        return value;
    value=calculate();
    return value;
}

@Fresh public X calculate(){
    counter++; //Fel
    return new X();
}

@Local public int count(){
    return counter++;
}
```

Exemplen visar några metoder med annoteringar mitt verktyg använder. Annoteringen *@Local* innebär ändring i samma objekt eller platser nåbara via *@Local* annoterade referenser. *@Pure* indikerar en metod som ska vara sidoeffektsfri förutom hemliga sidoeffekter till fält med *Secret*.

Memoriseringar använder sig av tillåtna sidoeffekter, nämligen att spara delresultaten i fält. Genom att göra dessa fält *@Secret* klarar analysen av att skilja på tillåtna och otillåtna sidoeffekter. Datagrupper kan vidare bestämma var memoriseringen få ske.