

Leon: sistem za verifikaciju softvera

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Milana Kovačević
mi14042@matf.bg.ac.rs

ma j 2018.

Abstract

Leon je sistem koji vrši statičku verifikaciju koda napisanog u programskom jeziku Scala. U okviru ovog rada prikazana je funkcionalnost sistema da odredi da li je program ispravan ili ne. Pored verifikacije funkcionalnog koda, Leon podržava i verifikaciju imperativnih programa. To vrši svodjenjem imperativnih programa na njima ekvivalentne, funkcionalne. Kako bi izvršio verifikaciju, generiše skup klauza na osnovu datog izvornog koda, a ispitivanje zadovoljivosti prosledjene formule vrši rešavač. Na osnovu rezultata rešavača, zaključuje o ispravnosti ulaznog programa. Za pronadjene neispravne delove koda, ovom metodom se dobija i kontra primer kojim se to potvrđuje. Korišćenje sistema Leon je opisano kroz primere, a date su i smernice za njegovo korišćenje.

Contents

1	Uvod	2
2	Jezik Leon	2
3	Algoritam	3
4	Rad sa imperativnim programima	4
5	Arhitektura sistema	5
6	Zaključak	6
	Literatura	6

1 Uvod

Programiranje u programskom jeziku Scala je u današnje vreme podržano od strane velikog broja razvojnih okruženja. U okviru njih, postoje alati koji olakšavaju programiranje vršeći leksičku i sintaksnu proveru napisanog koda. Sistem Leon pruža dodatnu pomoć: vrši statičku analizu ispravnosti napisanog koda u realnom vremenu! Pruža pouzdanu potvrdu ispravnosti, zasnovanu na proveru zadatih ograničenja za sve vrednosti koje se mogu naći u jednom izvršavanju. Kompletnost njegove provere pruža koristan alat koji ne propušta grešku u kodu. Ovo ga čini korisnim za debugovanje, kao i za proveru kritičnih delova koda.

Referentni rad se može naći na [4]. Više informacija o Leon sistemu, dokumentacija i primeri se mogu naći na njegovom zvaničnom sajtu [2].

2 Jezik Leon

Jezik koji sistem Leon zahteva na svom ulazu je jezik koji predstavlja podskup programskog jezika Scala [3]. Sastoji se od čisto funkcionalnog dela (eng. *PureScala*) i skupa podataka. Podržava većinu osnovnih osobina Skale: uparivanje šablona (eng. *pattern matching*), case grananja, složene tipove podataka. Ovaj podskup funkcionalnosti pruža dovoljno da korisnik može da definiše sopstvene strukture podataka i operacije nad njima. Specifikacije tih operacija se navode kroz konstrukcije: zahtevaj (eng. *require*) i osiguraj (eng. *ensuring*). Ove konstrukcije u osnovnoj Skali rade potpuno drugačije nego u Leonu. Skala ove provere sprovodi dinamički uz bacanje odgovarajućih izuzetaka, dok se moć Leon sistema nalazi u statičkom proveravanju tih specifikacija prilikom svih izvršavanja.

Primer verifikacije funkcije koja vrši sortiranje umetanjem (eng. *insertion sort*) je dat u primeru 2.1. Iako je Skala uglavnom funkcionalni programski jezik, ona podržava i neke imperativne konstrukcije. Leon je dovoljno pametan da zna da verifikuje i njih, a više o tome se nalazi u poglavlju 4.

Pored datog jednostavnog primera, Leon se dobro pokazao i u proveru složenijih algoritama, poput sortiranja spajanjem (eng. *merge sort*) ili brzog sort (eng. *quick sort*). Takodje, daje dobre rezultate i u proveru operacija nad veoma složenim strukturama podataka, kao što su operacije nad crveno-crnim stablima.

Primer 2.1 Sortiranje umetanjem.

Na primeru sortiranja umetanjem 1 ilustrovano je korišćenje konstrukcija zahtevaj i osiguraj. Podrazumeva se da su ranije definisane funkcije *isSorted(list)*, *size(list)* i *contents(element, list)*. Ove funkcije imaju jasnu svrhu: provera da li je lista sortirana, izračunavanje dužine liste i provera da li se element nalazi u listi. Prva funkcija koja vrši sortiranje umetanjem je ispravno napisana. Ukoliko se izostavi zahtev da ulazna lista bude sortirana, što je uradjeno u drugoj funkciji, Leon prijavljuje grešku i vraća korisniku kontra primer. Prikaz greške se može videti na slici 1.

```
def sortedIns(e: BigInt, l: List): List = {
  require(isSorted(l))
  l match {
    case Nil => Cons(e, Nil)
    case Cons(x, xs) => if (x <= e) Cons(x, sortedIns(e, xs))
                       else Cons(e, l)
  }
} ensuring(res => contents(res) == contents(l) ++ Set(e))
```

```

    && isSorted(res)
    && size(res) == size(l) + 1
  )
def buggySortedIns(e: BigInt, l: List): List = {
  l match {
    case Nil => Cons(e, Nil)
    Cons(x, xs) => if (x <= e) Cons(x, buggySortedIns(e, xs))
                  else Cons(e, l)
  }
} ensuring (res => contents(res) == contents(l) ++ Set(e)
    && isSorted(res)
    && size(res) == size(l) + 1
)

```

Listing 1: Sortiranje umetanjem.

Verification
×

Leon verifies the validity of all the verification conditions found in the selected function.

Invalid!

Function	Kind	Result	Time
buggySortedIns	postcondition	⚠ invalid	0.176
The following inputs violate the VC: <pre>e := BigInt(0) l := Cons(BigInt(1), Cons(BigInt(0), Cons(BigInt(0), Cons(BigInt(0), Nil))))</pre> It produced the following output: <pre>Cons(BigInt(0), Cons(BigInt(1), Cons(BigInt(0), Cons(BigInt(0), Cons(BigInt(0), Nil))))))</pre>			
buggySortedIns	match exhaustiveness	✓ valid	0.020

Repair

Close

Figure 1: Leon prijavljuje grešku i nalazi kontra primer u sortiranju umetanjem.

Leon se može isprobati na njegovoj zvaničnoj web stranici [2]. Na ovoj stranici se nalazi veliki skup detaljno opisanih primera, kao i tutorijali za upoznavanje sa sistemom. Pored toga, Leon je otvorenog koda, te se može isprobati i na lokalnoj mašini, kroz konzolnu aplikaciju. Kod se može naći na [1].

3 Algoritam

Statička verifikacija ispravnosti programa se izvršava proučavanjem izvornog koda, bez pokretanja programa. Algoritam koji koristi Leon se izvršava u dva koraka:

- Generisanje logičke formule na osnovu funkcionalnog izvornog koda;
- Ispitivanje zadovoljivosti dobijene logičke formule u kombinaciji sa mogućim vrednostima promenljivih.

Kako bi se došlo do logičke formule koja će kasnije biti prosledjena rešavaču, Leon svaku funkciju programa transformiše u ekvivalentan skup klauza. Na primer, funkcija koja izračunava dužinu liste:

```
def size(lst : List) : Int = lst match {
  case Nil => 0
  case Cons(_, xs) => 1 + size(xs)
}
```

se može predstaviti sledećim skupom klauza:

```
( size(lst) = e1 ) AND ( b1 <=> lst = Nil )
AND ( b1 ==> e1 = 0 ) AND ( ~b1 ==> e1 = 1 + size(lst.tail) )
```

Ova transformacija je intuitivna i predstavlja vezu između ulaza i izlaza funkcije. Glavna stvar na koju je potrebno obratiti pažnju prilikom ove transformacije je dodavanje dodatnih promenljivih kojima se prati tok izvršavanja funkcije. U ovom slučaju je to promenljiva *b1* čija vrednost *tačno* znači da se izvršila prvi slučaj u grananju (lista je prazna), a vrednost *netačno* znači da se izvršio drugi slučaj (lista nije prazna).

Na ovu generisanu formulu se iterativno dodaju klauze koje sadrže ograničenja formirana na osnovu mogućih vrednosti promenljivih u kodu. Nakon toga se ispituje zadovoljivost cele formule korišćenjem SMT rešavača. Ukoliko je formula nezadovoljiva, dodato ograničenje predstavlja kontraprimer. Inače, u sledećoj iteraciji si ispituje neko drugo ograničenje, tj. ulaz. Ova procedura je polu-odlučiva, što znači da ne garantuje da će se svaki put završiti. Zato Leon ima upapred definisano vremensko ograničenje tokom koga će vršiti proveru. Medjutim, ima teorijski zasnovanu osobinu da će uvek naći kontraprimer ukoliko on postoji.

4 Rad sa imperativnim programima

Leon može da verifikuje i imperativne programe. Prvo ih transformiše u funkcionalne, a zatim vrši verifikaciju novog transformisanog programa. Početni i transformisani program su ekvivalentni, te se dokazivanjem ispravnosti transformisanog programa posredno dokazuje i ispravnost početnog programa.

Transformacija se vrši tako što se petlja prevede u ugnježdenu repnu rekurzivnu funkciju. Transformacija programa je rekurzivan postupak koji menja imperativne izraze u niz definicija (*val* i *def*) koje uvode nova imena i promenljive. Kako bi se sačuvala informacija o početnim promenljivama, one se mapiraju i njihova vrednost se pamti preko izvedenih promenljivih. Na primer, sledeći izraz:

```
x=2
y=3
x=y+1
```

se može transformisati u ekvivalentni izraz:

```
val x1 = 2
val y1 = 3
val x2 = y1 + 1
x = x2
y = y1
```

Svodjenje petlje u njen rekurzivan oblik postaje lako kada se prvo ovako transformisu izrazi. Invarijanta petlje prelazi u preduslove i postuslove rekurzivne funkcije. Na primeru 4.1 je prikazan program koji izračunava ceo deo korena broja n , a nakon toga je navedena njegova transformacija.

Primer 4.1 Ceo deo korena broja n .

```
def sqrt(n : Int) : Int = {
  var toSub = 1
  var left = n
  while( left >= 0) {
    if(toSub % 2 == 1)
      left -= toSub
    toSub += 1
  }
  (toSub / 2) - 1
}
```

Listing 2: Ceo deo korena broja n .

```
def sqrt(n : Int) : Int = {
  val toSub4 = 1
  val left5 = n
  def rec(left3 : Int, toSub2 : Int) = if(left3 >= 0) {
    val left2 =
      if(toSub3 % 2 == 1) {
        val left1 = left3 - toSub2
        left1
      } else {
        left3
      }
    val toSub1 = toSub2 + 1
    rec(left2, toSub1)
  } else {
    (left3, toSub2)
  }
  val (left4, toSub3) = rec(left5, toSub4)
  (toSub3 / 2) - 1
}
```

Listing 3: Transformisani program.

Formalno definisanje ovih transformacija je van domena ovog rada. Više o tome se može naći u referentnom radu [4].

5 Arhitektura sistema

Prikaz arhitekture Leon sistema se nalazi na slici 2. Arhitektura sistema se sastoji od više međusobno povezanih celina:

- *Front end* - Koristi se u ranim fazama kompajliranja Skala koda. Leon presreće određenu fazu kompilacije, filtrira ulaz na podržani i formira posebno sintaksno stablo koje on zna da čita. Na ovaj način se oslanja na već postojeću implementaciju koja radi parsiranje koda, kao i određivanje i proveru tipova promenljivih.
- *Rešavač* - Rešavač se u potpunosti oslanja na Z3 SMT rešavač. Njemu se putem interfejsa prosledjuju definisana ograničenja.
- *Generator koda* - Izračunava istinitosnu vrednost terma.

- *Kontroler kraja izvršavanja* - Kontroler vrši proveru da li se ulazni kod završava. Ovo je opšte poznati problem [5], ali Leon ga uprošćava i tvrdi da se program završava ukoliko se parametri u rekurzivnim pozivima smanjuju. Ova komponenta je i dalje u razvoju, a nije ni postojala u ranijim verzijama sistema.
- *Web interfejs* - Korišćenjem web interfejsa [2] je najlakše doći u susret sa Leon sistemom. U okviru njega se nalazi tekst editor unutar koga se periodično poziva Leon. Na taj način se u realnom vremenu proverava ispravnost napisanog koda.

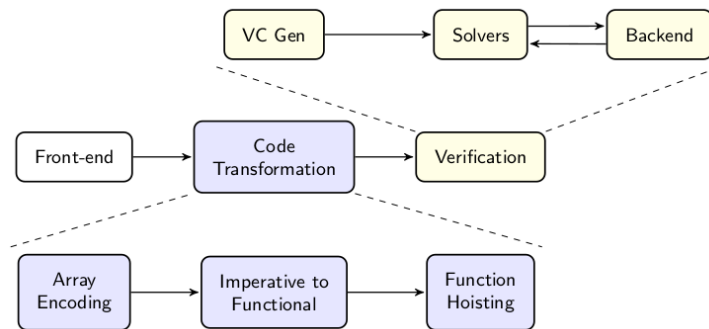


Figure 2: Arhitektura sistema.

6 Zaključak

Sistem Leon je i dalje u razvoju. Vremenom su dodate nove funkcionalnosti, kao što je, na primer, ispravljanje prethodno nađenog neispravnog koda. Iako je sistem napravljen za programski jezik Skala, on sadrži opšte koncepte koji se mogu primeniti i nad ostalim programskim jezicima, kako funkcionalnim, tako i imperativnim. Ovaj sistem je upravo zbog toga od velikog teorijskog značaja, a njegova integracija u neko od poznatih razvojnih okruženja bi mu povećala popularnost i doprinelo njegovoj široj upotrebi.

References

- [1] Leon Github Repositorz. on-line at: <https://github.com/epfl-lara/leon>.
- [2] Leon official website. on-line at: <http://lara.epfl.ch/leon/>.
- [3] Scala Programming Language. on-line at: <https://www.scala-lang.org/>.
- [4] Etienne Kneuss Régis Blanc, Viktor Kuncak and Philippe Suter. An overview of the leon verification system: verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*, 2013.
- [5] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.