Formalna verifikacija programa

Andrej Ivašković

Matematička gimnazija

NEDELJA$^4_{\text{INFORMATIKE}}$

26. mart 2018.

Kako možemo da garantujemo tačnost programa?

- Ne bi bilo loše kada bi programi koje pišemo zapravo radili kako treba (akcenat je na tačnosti, ne efikasnosti).
- Izvori bagova su razni, težine detekcija i ispravljanja su raznovrsne (konkurentni programi su posebno problematični).
- Veći broj ovih problema bi trebalo primetiti i ispraviti pre nego što softver bude na raspologanju ciljnim korisnicima.
- **Code review**: drugi članovi tima će analizirati preporučene promene koda.
- **Testiranje**?

Testiranje

- Srećom, danas su nam na raspolaganju alati i paketi za automatsko testiranje softvera.
- **Test driven development (TDD)**: testovi se pišu pre implementacije.
- **Unit testing**: testiranje malih modula (često pojedinačnih funkcija), uz pokrivanje svih relevantnih slučajeva.
- **Integration testing**: testiranje interakcije nekoliko modula.
- **Regression testing**: baza testova ostaje ista i bilo koja promena koda zahteva pokretanje svih relevantnih testova.
- Postoje alati za analizu **pokrivenosti** skupa testova (koji analiziraju koji delovi koda su bili obuhvaćeni testiranjem – na primer, koja grana `IF` naredbe).

Nedostaci testiranja

- Predvideti sve slučajeve i načine upotrebe nekog modula je nemoguće.
- Nekada je implementacija takva da namenski prolaze skoro isključivo jednostavni unit testovi.
- Testovi (na osnovu kojih je pisana implementacija) mogu da imaju netačne rezultate.
- Neki problemi nestaju pri testiranju ili program može da poseduje izvesnu dozu nedeterminizma.
- **Testovi mogu da dokažu prisustvo, ali ne i odsustvo bagova.** Ključno je da počinjemo da razmišljamo o **dokazima korektnosti**.

Primeri katastrofa

- Postoje neki scenariji u kojima upotreba softvera koji nije u potpunosti tačan može da ima katastrofalne posledice.
- U medicini: Therac-25 mašina za radioterapiju je izazvala nekoliko smrti ili trajnih povreda (bag u sinhronizaciji paralelnih procesa).
- U odbrani: 1991. u toku Zalivskog rata, Patriot antibalistička raketa ne uspeva da presretne iračku raketu, koja potom pogađa američku bazu u Saudijskoj Arabiji (bag usled floating-point aritmetike).
- U nauci: ESA lansira Ariane 5 raketu 1996. godine, do eksplozije dolazi nakon 37 sekundi (bag nastao usled promene koda u odnosu na Ariane 4).

Ariane 5 eksplozija

goto fail

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                         uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

   if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
   if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
   if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

Formalna semantika

- Za **formalno rezonovanje** o programima je neophodno razviti formalan model ponašanja (semantike) programa.
- Postoje tri osnovna pristupa **formalnoj semantici programskih jezika**:
  - **Operaciona.** Posmatraju se **konfiguracije** oblika $\langle e, s \rangle$, gde je e izraz u nekom programskom jeziku, a s stanje ili okruženje (vrednosti promenljivih), kao i prelazi $\langle e_1, s_1 \rangle \to \langle e_2, s_2 \rangle$ između tih konfiguracija.
  - **Denotaciona.** Pri nekom stanju s se izrazu e dodeljuje značenje koje odgovara nekom matematičkom objektu $[\![e]\!]_s$ (na primer, brojevima u programskom jeziku odgovaraju brojevi, potprogramima parcijalne funkcije).
  - **Aksiomatska.** Eksplicitno se dokazuju svojstva programa tako što se generišu određeni **dovoljni uslovi** vođeni strukturom programa, nakon čega se koristi automatski dokazivač teorema za kompletiranje radi konstrukcije formalnog dokaza.

Hoare logika

- **Hoare logika** je aksiomatski sistem koji je razvio C. A. R. Hoare (**Tony Hoare**) sa Oksforda.
- Verovatno najuspešniji i najpraktičniji pristup formalnoj semantici i verifikaciji softvera.
- Sistem je zasnovan na **preduslovima** i **postuslovima**, sa ključnim konceptom **Hoare trojke** $\{P\}\ C\ \{Q\}$.
- Mi ćemo posmatrati osnovnu varijantu koja omogućava rezonovanje o najprostijim programskim jezicima i strukturama.

Parcijalna i totalna korektnost

- Napisati $\{P\}\ C\ \{Q\}$ znači da, ako važi preduslov P, tada komanda/program C ili rezultuje beskonačnom petljom ili se završava, pri čemu po izvršavanju važi Q. Dokaz ovakvog tvrđenja je dokaz **parcijalne korektnosti**.
- Intuitivan primer: $\{\mathtt{X} = 2\}\ \mathtt{X}\ \mathtt{:=}\ 3\ \{\mathtt{X} = 3\}$
- $[P]\ C\ [Q]$ je jače tvrđenje od $\{P\}\ C\ \{Q\}$, koje garantuje da će C završiti izvršavanje. Dokaz ovakvog tvrđenja je dokaz **totalne korektnosti**.
- Fokus će nam biti na parcijalnoj korektnosti.
- Najpre ćemo pogledati primere za koje intuitivno znamo da su tačni.

Primeri

$$\{\mathtt{X} = 1 \wedge \mathtt{Y} = 2\}$$
$$\mathtt{Z := X;\ X := Y;\ Y := Z}$$
$$\{\mathtt{X} = 2 \wedge \mathtt{Y} = 1\}$$

$$\{\mathtt{X} = 1\}$$
$$\mathtt{IF\ X = 1\ THEN\ Y := 10\ ELSE\ Y := 20}$$
$$\{\mathtt{X} = 1 \wedge \mathtt{Y} = 10\}$$

$$\{\mathtt{X} = 1\}$$
$$\mathtt{IF\ X = 1\ THEN\ Y := 10\ ELSE\ Y := 20}$$
$$\{\mathtt{X} = 1 \wedge \mathtt{Y} = 20\}$$

Primeri

$$\{\mathtt{X} = 1 \wedge \mathtt{X} = 2\}$$
```
X := 3
```
$$\{\mathtt{X} = 42\}$$

$$\{\mathtt{X} = 10\}$$
```
S := 0; Y := X;
WHILE Y != 0 DO (S := S + X; Y := Y - 1)
```
$$\{\mathtt{X} = 10 \wedge \mathtt{S} = \mathtt{X}(\mathtt{X}+1)/2\}$$

$$\{\mathtt{X} = -5\}$$
```
S := 0; Y := X;
WHILE Y != 0 DO (S := S + X; Y := Y - 1)
```
$$\{\mathtt{X} = -5 \wedge \mathtt{S} = \mathtt{X}(\mathtt{X}+1)/2\}$$

Verifikacija TimSorta

- Detekcija baga u TimSort implementaciji je jasan praktičan rezultat formalnih metoda – Gouw et al. *OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case*.
- Prilikom pokušaja formalne verifikacije nekih biblioteka pisanih u Javi korišćenjem KeY lociran je, nakon mnogo godina, bag u ugrađenom, *default* TimSort algoritmu (rezultat baga je bio `ArrayOutOfBoundsException`).
- Konstrukcija problematičnog ulaza nije bila jednostavna.
- Bag je lociran u funkciji `mergeCollapse()`.

TimSort bag

```
private void mergeCollapse() {
     while (stackSize > 1) {
         int n = stackSize - 2;
       if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1]) {
             if (runLen[n - 1] < runLen[n + 1])
                 n--;
             mergeAt(n);
         } else if (runLen[n] <= runLen[n + 1]) {
             mergeAt(n);
         } else {
             break; // Invariant is established
         }
     }
 }
```

Ispravka TimSort baga

```
private void newMergeCollapse() {
  while (stackSize > 1) {
    int n = stackSize - 2;
   if (   (n >= 1 && runLen[n-1] <= runLen[n] + runLen[n+1])
      || (n >= 2 && runLen[n-2] <= runLen[n] + runLen[n-1])) {
            if (runLen[n - 1] < runLen[n + 1])
                n--;
          } else if (runLen[n] > runLen[n + 1]) {
              break; // Invariant is established
          }
          mergeAt(n);
      }
 }
```

Reakcija na detektovan bag

- Autori rada su javili Java developerima gde se nalazi problem i predložili izmenu koda.
- Međutim, jednostavnija ispravka (izmena predefinisane konstante) je bilo rešenje koje su odabrali autori originalnog koda, radi jednostavnosti.
- Međutim, ova ispravka je memorijski neefikasna.
- Verovatan razlog: nerazumevanje šta su autori hteli da kažu.

Svojstva aksiomatskih sistema

- Često se uz aksiomatsku semantiku definiše i operaciona.
- Kažemo da je aksiomatski sistem **ispravan** ukoliko je svaka dokaziva Hoare trojka $\{P\}\ C\ \{Q\}$ takođe i tačna.
- Kažemo da je aksiomatski sistem **kompletan** ukoliko bilo koja Hoare trojka $\{P\}\ C\ \{Q\}$ koja je tačna takođe može i da se dokaže.
- Kompletnost skoro nikada ne važi, a ispravnost je ključna.

Digresija: specifikacija

- Možemo da posmatramo i zadatak $\{P\}\ ?\ \{Q\}$: odrediti program C takav da za date P i Q važi $\{P\}\ C\ \{Q\}$, ili ustanoviti da ne postoji.
- Ovo je problem **specifikacije**.
- Automatski naći odgovor je daleko teži zadatak!

Nedostaci i glavna prednost aksiomatskih sistema

- Neophodno je napisati preduslove, postuslove, kao i dati izvesnu dozu pomoći automatskom dokazivaču teorema (videćemo kasnije).
- Pisanje preduslova i postuslova zahteva iskustvo i vežbu, kao i drugačiji način razmišljanja o programima.
- S druge strane, dovoljno je *jednom* dokazati da modul radi po želji.

Induktivno definisana pravila

- Glavna ideja je da dođemo do jasnog metoda dokazivanja $\{P\}\ C\ \{Q\}$ za neke P, C, Q.
- Osnovna ideja je da se definišu dovoljni (ne potrebni) uslovi da bi se dokazale Hoare trojke, u zavisnosti od strukture C.
- Neki od ovih uslova zahtevaju da se dokaže $\{P'\}\ C'\ \{Q'\}$, gde je C' deo C.
- Generisanje svih potrebnih uslova za $\{P\}\ C\ \{Q\}$ može da se uradi onda rekurzivno (induktivno).
- P i Q su napisani jezikom klasične logike prvog reda i aritmetike celih brojeva, a C je pisan u programskom jeziku.

Pregled WHILE jezika

- Posmatramo WHILE jezik radi demonstracije Hoare pravila: jezik sa logičkim i celobrojnim tipovima, pri čemu su sve promenljive V celobrojne.
- Razmatramo tri klase izraza u WHILE jeziku:
  - **komande C**: C_1 ; C_2 | IF B THEN C_1 ELSE C_2 | V := E | SKIP | WHILE B DO C_1 | ...
  - **aritmetički izrazi E**: E_1 + E_2 | E_1 - E_2 | E_1 * E_2 | ...
  - **logički izrazi B**: E_1 = E_2 | E_1 != E_2 | E_1 > E_2 | ...
- Logički i aritmetički izrazi su u ovom slučaju podskup jezika kojim pišemo preduslove i postuslove.

IF pravilo

- Za dokazivanje $\{P\}$ `IF` B `THEN` C_1 `ELSE` C_2 $\{Q\}$ je dovoljno dokazati:
  - $\{P \wedge B\}\ C_1\ \{Q\}$
  - $\{P \wedge \neg B\}\ C_2\ \{Q\}$
- Dakle, dokazati:

$$\{\texttt{X} = 1\}$$
`IF X = 1 THEN Y := 10 ELSE Y := 20`
$$\{\texttt{X} = 1 \wedge \texttt{Y} = 10\}$$

značiti dokazati naredna dva tvrđenja:

- $\{\texttt{X} = 1 \wedge \texttt{X} = 1\}$ `Y := 10` $\{\texttt{X} = 1 \wedge \texttt{Y} = 10\}$
- $\{\texttt{X} = 1 \wedge \neg(\texttt{X} = 1)\}$ `Y := 20` $\{\texttt{X} = 1 \wedge \texttt{Y} = 10\}$

Naredba dodele

- Možda bi bilo očekivano da je za dokazivanje $\{P\}\ V\ :=\ E\ \{Q\}$ dovoljno pokazati $P[E/V] \Rightarrow Q$, gde $P[E/V]$ znači zameniti sva pojavljivanja V u P sa E. **Međutim, ovo pravilo nije ispravno!**
- Umesto toga, za dokazivanje $\{P\}\ V\ :=\ E\ \{Q\}$ je dovoljno dokazati $P \Rightarrow Q[E/V]$.
- Dokazati $\{\mathtt{X} = 1\}\ \mathtt{Y}\ :=\ \mathtt{10}\ \{\mathtt{X} = 1 \wedge \mathtt{Y} = 10\}$ znači da je dovoljno pokazati $\mathtt{X} = 1 \wedge 10 = 10 \Rightarrow \mathtt{X} = 1$.
- Dokazati $\{\mathtt{X} = 1\}\ \mathtt{X}\ :=\ \mathtt{X}\ +\ 1\ \{\mathtt{X} = 2\}$ znači da je dovoljno pokazati $\mathtt{X}\ +\ 1 = 2 \Rightarrow \mathtt{X} = 1$.

Pravilo vezivanja

- Kada imamo više instrukcija koje slede jedna za drugom, reč je o instrukciji oblika $C_1\,;\;\; C_2$.
- Za dokazivanje $\{P\}\; C_1\,;\;\; C_2\; \{Q\}$ je dovoljno odrediti neko R takvo da je moguće dokazati:
  - $\{P\}\; C_1\; \{R\}$
  - $\{R\}\; C_2\; \{Q\}$
- Na primer, za dokazivanje

$$\{\top\}\ \mathsf{X}\ :=\ 1\,;\ \ \mathsf{Y}\ :=\ 2\ \{\mathsf{X} = 1 \wedge \mathsf{Y} = 2\}$$

  dovoljno je posmatrati $R \equiv \mathsf{X} = 1$, i dokazati:
  - $\{\top\}\ \mathsf{X}\ :=\ 1\ \{\mathsf{X} = 1\}$
  - $\{\mathsf{X} = 1\}\ \mathsf{Y}\ :=\ 2\ \{\mathsf{X} = 1 \wedge \mathsf{Y} = 2\}$
- Kako uopšte dolazimo do ovog R (automatski)?

Najopštiji preduslov

- Posmatrajmo ukratko malo drugačiji problem: odrediti najopštije P takvo da za date C i Q važi $\{P\}\ C\ \{Q\}$. Ovo ćemo označiti sa $\mathbf{wpc}(C, Q)$.
- Na osnovu prethodnih pravila nije teško pokazati:
  - $\mathbf{wpc}(V\ :=\ E, Q) = Q[E/V]$
  - $\mathbf{wpc}(C_1;\ C_2, Q) = \mathbf{wpc}(C_1, \mathbf{wpc}(C_2, Q))$
  - $\mathbf{wpc}(\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2, Q) =$ $(B \Rightarrow \mathbf{wpc}(C_1, Q)) \wedge (\neg B \Rightarrow \mathbf{wpc}(C_2, Q))$
- Dakle, za dokazivanje $\{P\}\ C\ \{Q\}$ je dovoljno pokazati $P \Rightarrow \mathbf{wpc}(C, Q)$, i prethodno navedena pravila zaključivanja za sada nije neophodno koristiti.

Ulazni podaci

- Pretpostavimo da želimo da izrazimo jezikom Hoare trojki da neki program računa maksimum promenljivih X i Y i zapisuje ga u promenljivu Z.
- Međutim, uslov $\{\top\}\ C\ \{\mathsf{Z} \geq \mathsf{X} \wedge \mathsf{Z} \geq \mathsf{Y} \wedge (\mathsf{Z} = \mathsf{X} \vee \mathsf{Z} = \mathsf{Y})\}$ je problematičan, budući da naredni program C zadovoljava ovu trojku:

$$C \equiv \mathsf{X}\ :=\ 1;\ \ \mathsf{Y}\ :=\ 1;\ \ \mathsf{Z}\ :=\ 1$$

- Uvodi se **konvencija** da promenljive čije ime počinje malim slovom predstavljaju ulazne podatke kojima program nema nikakav pristup.
- Zato tražimo trojku:

$$\{\mathsf{X} = \mathsf{x} \wedge \mathsf{Y} = \mathsf{y}\}\ C\ \{\mathsf{Z} \geq \mathsf{x} \wedge \mathsf{Z} \geq \mathsf{y} \wedge (\mathsf{Z} = \mathsf{x} \vee \mathsf{Z} = \mathsf{y})\}$$

Primer punog dokaza

$$\{X = x \wedge Y = y\}$$
$$\texttt{IF X > Y THEN Z := X ELSE Z := Y}$$
$$\{Z \geq x \wedge Z \geq y \wedge (Z = x \vee Z = y)\}$$

Primer punog dokaza

$$\{X = x \wedge Y = y\}$$
$$\texttt{IF X > Y THEN Z := X ELSE Z := Y}$$
$$\{Z \geq x \wedge Z \geq y \wedge (Z = x \vee Z = y)\}$$

Dovoljno je pokazati:

$$\begin{aligned}(X = x \wedge Y = y) \quad &\Rightarrow \quad ((X > Y \Rightarrow (X \geq x \wedge X \geq y \wedge (X = x \vee X = y))) \\ &\wedge \quad (\neg(X > Y) \Rightarrow (Y \geq x \wedge Y \geq y \wedge (Y = x \vee Y = y))))\end{aligned}$$

što sledi iz celobrojne aritmetike.

Problem sa WHILE ciklusom

- Pojava ciklusa u jeziku je ono što dovodi do problema i tu ponašanje programa postaje manje predvidljivo.
- Jedan od problema je, između ostalog, što sada postoji mogućnost da se ciklus neće nikada završiti.
- Primetimo da su naredna dva programa semantički ekvivalentna:

$$\texttt{WHILE } B \texttt{ DO } C$$

$$\texttt{IF } B \texttt{ THEN } C \texttt{ ELSE SKIP; WHILE } B \texttt{ DO } C$$

- To znači da, ako dokazujemo $\{P\}\ \texttt{WHILE } B \texttt{ DO } C\ \{Q\}$, možemo da tražimo neko R takvo da važe:
  - $\{P\}\ \texttt{IF } B \texttt{ THEN } C \texttt{ ELSE SKIP}\ \{R\}$
  - $\{R\}\ \texttt{WHILE } B \texttt{ DO } C\ \{Q\}$
- Međutim, ovaj postupak možemo da ponovimo beskonačno mnogo puta!

Korišćenje invarijanti

- Dovoljno je osmisliti uslov I, **invarijantu ciklusa**, koja važi u svakoj iteraciji ciklusa (uključujući i neposredno pre i neposredno posle).
- Dakle, dovoljno je odrediti takvo I da važe sledeći uslovi:
  - $P \Rightarrow I$
  - $I \wedge \neg B \Rightarrow Q$
  - $\{I \wedge B\}\ C\ \{I\}$
- Ovu invarijantu mora da smisli programer (odnosno čovek koji priprema program za automatsku verifikaciju) i ovo nije nešto do čega može automatski da se dođe!

Primer dokaza sa WHILE ciklusom

$\{\mathtt{X} = \mathtt{x} \wedge \mathtt{X} \geq 0\}$

```
S := 0;
WHILE X > 0 DO (S := S + X; X := X - 1)
```

$\{\mathtt{S} = \mathtt{x}(\mathtt{x}+1)/2\}$

Primer dokaza sa WHILE ciklusom

$\{\mathtt{X} = \mathtt{x} \wedge \mathtt{X} \geq 0\}$
`S := 0;`
$\{\mathtt{X} = \mathtt{x} \wedge \mathtt{X} \geq 0 \wedge \mathtt{S} = 0\}$
`WHILE X > 0 DO (S := S + X; X := X - 1)`
$\{\mathtt{S} = \mathtt{x}(\mathtt{x}+1)/2\}$

Koju invarijantu izabrati?

Primer dokaza sa WHILE ciklusom

$\{\mathtt{X} = \mathtt{x} \wedge \mathtt{X} \geq 0\}$
`S := 0;`
$\{\mathtt{X} = \mathtt{x} \wedge \mathtt{X} \geq 0 \wedge \mathtt{S} = 0\}$
`WHILE X > 0 DO (S := S + X; X := X - 1)`
$\{\mathtt{S} = \mathtt{x}(\mathtt{x}+1)/2\}$

Koju invarijantu izabrati?
Odabirom $I \equiv \mathtt{S} = \mathtt{x}(\mathtt{x}+1)/2 - \mathtt{X}(\mathtt{X}+1)/2 \wedge \mathtt{X} \geq 0$ dolazimo do toga da treba dokazati:

- $\mathtt{X} = \mathtt{x} \wedge \mathtt{X} \geq 0 \Rightarrow I$
- $I \wedge \neg(\mathtt{X} > 0) \Rightarrow Q$
- $\{I \wedge \mathtt{X} > 0\}$ `S := S + X; X := X - 1` $\{I\}$

Kako se ovo dokazuje?

Aksiomatske metode u praksi

- Pre verifikacije, od programera se očekuju anotacije početnog preduslova, krajnjeg postuslova i invarijanti svih ciklusa – svi ostali uslovi mogu da se odrede primenom $\mathbf{wpc}$ metoda.
- Invarijante su najkomplikovanije: ako u jeziku postoje nizovi, često se pojavljuju kvantifikatori $\forall$ i $\exists$, te više nije dovoljno rezonovati samo o aritmetici i logici iskaza.
- Why3 sistem za verifikaciju koristi ML-ovski jezik i može da se koristi za verifikaciju algoritama implementiranih u uprošćenom funkcionalnom stilu.
- KeY projekat je sistem za Javu koji se oslanja na jezik JML za specifikaciju, korišćen u prethodno navedenom TimSort primeru.

KeY i JML

```
/*@ requires
  @   stackSize > 0 &&
  @  runLen[stackSize-4] > runLen[stackSize-3]+runLen[stackSize-2]
  @   && runLen[stackSize-3] > runLen[stackSize-2];
  @ 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()

.................................

/*@ loop_invariant
  @  (\forall int i; 0<=i && i<stackSize-4;
  @              runLen[i] > runLen[i+1] + runLen[i+2])
  @  && runLen[stackSize-4] > runLen[stackSize-3])
  @*/
```

Mogućnosti Hoare logike

- Videli smo kako Hoare logika može da se koristi za verifikaciju jednostavnih programa.
- Neophodna je ručna anotacija preduslova i postuslova, kao i invarijanti ciklusa, pre nego što se iskoristi automatski dokazivač teorema koji može da rezonuje o aritmetici.
- Nismo pokazali da su pravila koja smo demonstrirali ispravna – za ovo se ispostavlja da nije teško, ali jeste naporno.

Dinamički podaci

- Nismo se bavili pokazivačkim i referentnim tipovima, odnosno dinamički alociranim podacima.
- Osnovna Hoare logika ne može da se bavi ovom analizom, ali postoji proširenje: *separation logic*.
- Ovde su pravila zaključivanja nešto komplikovanija, ali imaju jasne praktične rezultate.

Problemi sa paralelnim procesima

- Paralelni procesi predstavljaju izvor problema mnogim ljudima i možemo da se susretnemo sa bagovima razne vrste.
- Postoji nekoliko aksiomatskih sistema za rezonovanje o paralelnim procesima, uključujući:
  - Owicki-Gries logic
  - rely-guarantee reasoning
  - concurrent separation logic

Tipovi i garancije

- Alternativa je da promenimo programski jezik i programsku paradigmu.
- Savremeni (ali često ne mnogo popularni) funkcionalni jezici imaju složene sisteme tipova koji najveći broj bagova pretvaraju u greške pri kompajliranju.
- Ipak, ovi programski jezici često zahtevaju još više matematičkog znanja i poznavanja teorijskog računarstva u odnosu na korišćenje automatskih dokazivača teorema.

$$\{P\}\ C\ \{Q\}$$