Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/17BE21Jokic

TRANSFORMACIJE STABALA APSTRAKTNE SINTAKSE U PROGRAMSKI KOD

TRANSFORMATIONS OF ABSTRACT SYNTAX TREES INTO SOURCE CODE

Andrej Jokić, *Fakultet tehničkih nauka, Novi Sad*

Oblast – RAČUNARSTVO I AUTOMATIKA

Kratak sadržaj – *Mnogi napredni alati koji automatizuju izmjene programskog koda, kao što su alati za automatsko refaktorisanje, zasnivaju se na izmjenama stabla apstraktne sintakse. Izmjene izvršene na stablu je zatim potrebno primijeniti na originalni izvorni kod, što se može postići transformacijom stabla nazad u tekstualni oblik. U ovom radu su istražena i implementirana rješenja za implementaciju ovakvih transformacija, uz očuvanje nesemantičkih elemenata originalnog izvornog koda, kao što su komentari i ručno formatiranje.*

Ključne reči: *Stablo apstraktne sintakse, jezici specifični za domen, refaktorisanje*

Abstract – *Most of the advanced tools for automated source code modifications, such as automated refactoring tools, are based on modifications of the abstract syntax tree. The modifications which are done on the abstract syntax tree must be applied to the original source code, which can be done by transforming the tree back into a textual representation. In this paper, solutions for such transformations are researched and implemented, focusing on the problems of preserving the non-semantic elements of the original source code, such as comments and manual formatting.*

Keywords: *Abstract syntax tree, domain specific languages, refactoring*

1. UVOD

Razvoj računarskih tehnologija napreduje velikom brzinom, pružajući korisnicima sve više softverskih funkcionalnosti. Obim poslova koji se obavljaju na računaru je iz godine u godinu sve veći, a samim tim rastu mogućnosti i kompleksnost softvera. Pri tome, zahtjevi klijenata se veoma često mijenjaju pa je potrebno neprekidno održavati i nadograđivati softver. Uz porast kompleksnosti softvera raste i cijena njegovog razvoja i održavanja. Zbog toga, dosta pažnje se posvećuje istraživanju i razvoju tehnologija koje programerima olakšavaju posao i povećavaju produktivnost.

U savremenom programiranju koristi se mnoštvo alata koji automatizuju mnoge radnje koje programeri često obavljaju.

Jedan primjer ovakvih alata su alati za automatsko refaktorisanje koda. Refaktorisanje podrazumijeva izmjene programskog koda radi poboljšanja njegove interne strukture, ali bez promjene spoljašnjeg ponašanja koda. Naime, neprestane izmjene i nadogradnje softvera dovode do povećanja kompleksnosti i zamršenosti koda, što ga vremenom čini sve težim za održavanje i proširivanje. Da bi se kod održavao kvalitetnim i čistim, potrebno je neprestano refaktorisati djelove koda koji mogu biti implementirani na jednostavniji i jasniji način [1].

Za razvoj preciznih alata za refaktorisanje neophodno je da alati vrše dublju analizu izvornog koda i da izmjene vrše uzimajući u obzir semantiku elemenata koda. Zbog toga, mnogi napredni alati za refaktorisanje ne vrše izmjene direktno nad tekstualnim oblikom koda, već za to koriste neku vrstu apstraktne reprezentacije koda. Stablo apstraktne sintakse (eng. Abstract Syntax Tree, skraćeno AST) je konačno usmjereno stablo koje sadrži samo suštinu semantike programa i dobija se iz izvornog koda postupkom koji se zove parsiranje ili sintaksna analiza. Stablo apstraktne sintakse je veoma jednostavno za analizu od strane računarskih programa i omogućava kreiranje veoma preciznih alata za automatske izmjene izvornog koda.

Nakon izvršenih izmjena na stablu apstraktne sintakse, potrebno je date izmjene primijeniti i na originalni izvorni kod. Ovo se može uraditi postupkom koji je suprotan od parsiranja, tj. transformisanjem stabla apstraktne sintakse nazad u tekstualni oblik. Međutim, kako stablo apstakne sintakse sadrži samo suštinu značenja programa, mnogi elementi koda koji nemaju semantičkog značaja se gube tokom parsiranja, pa stablo ne sadrži sve informacije koje su potrebne za rekonstrukciju originalnog izvornog koda. Zbog toga, implementacija ovakvih transformacija nije jednostavna i zahtijeva pokrivanje mnogih graničnih slučajeva kako bi se dobio kvalitetan rezultat [2].

Cilj ovog rada jeste implementacija rješenja za transformaciju stabala apstraktne sintakse u programski kod u okviru alata za razvoj jezika specifičnih za domen TextX. U radu će biti istraženi problemi koji se susrijeću prilikom implementacije ovakvog postupka, kao i moguća rješenja datih problema. Na kraju će biti predstavljena implementacija rješenja i prikaz dobijenih rezultata.

2. PROBLEMI

2.1. Ključne riječi

Prvi problem na koji se nailazi pri generisanju teksta iz stabla apstraktne sintakse je nedostatak nekih elemenata izvornog koda. Zagrade, znaci interpunkcije i ključne riječi ne doprinose semantici programa već služe za prepoznavanje različitih struktura u kodu prilikom parsiranju kao i za bolju čitljivost koda. Prema tome, ovi

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Igor Dejanović, vanr. prof.

elementi su neophodni za generisanje ispravnog koda i potrebno je pronaći način da budu sačuvani.

Jedno rješenje ovog problema podrazumijeva čuvanje datih elemenata u samom stablu apstraktne sintakse. Za ovo je potrebno modifikovati parser tako da ove elemente takođe uključi u stablo. Kako ne bi bila narušena struktura stabla, i kako bi ostali programi koji koriste ovo stablo nastavili da rade kako je očekivano, dodatne podatke je poželjno sačuvati kao atribute postojećih čvorova stabla. Kada su svi neophodni elementi sačuvani na ovaj način, generisanje izvornog koda se svodi na prolazak kroz svaki čvor stabla redom i štampanje sačuvanih podataka [2].

Alternativni pristup se ne oslanja na čuvanje podataka u stablu, već na generisanje nedostajućih podataka prilikom transformacije stabla u tekst. Naime, ukoliko je poznat oblik različitih struktura u jeziku, na osnovu podataka iz stabla apstraktne sintakse mogu se generisati elementi koji su neophodni da bi rezultujući kod odgovarao sintaksi jezika. Prednost ovog pristupa je što nije potrebno modifikovati postojeći parser i dodavati podatke u stablo, ali je sa druge strane potrebno pisati logiku za štampanje svakog elementa jezika [3].

2.2. Formatiranje

Formatiranje se odnosi na vizuelnu strukturu i izgled koda. Dok formatiranje nije neophodno za generisanje ispravnog koda, ono je veoma značajno za preglednost i čitljivost. Vizuelnu strukturu koda najčešće definišu prazni karakteri - razmaci, prelomi redova i prazni redovi. Slično kao kod ključnih riječi, prazni karakteri se mogu dobiti na dva osnovna načina, čuvanjem dodatnih atributa u stablu ili generisanjem novog formatiranja. Međutim, u ovom slučaju je potrebno obratiti pažnju na još neke slučajeve.

Ukoliko se prazni karakteri čuvaju u stablu, bitno je obratiti pažnju kom čvoru se dodjeljuju prazni karakteri, u suprotnom može doći do toga da refaktorisanjem elemenata stabla prazni karakteri završe na pogrešnom mjestu. Formatiranje nije formalno definisano u jeziku već programer ima slobodu da ga koristi kako želi, pa je veoma teško definisati pravila za tretiranje formatiranja, i potrebno je uvoditi određene pretpostavke [4].

Sa druge strane, generisanje formatiranja, poznato pod nazivom pretty-printing, takođe može proizvesti dobre rezultate. Međutim, ponovno generisanje formatiranja predstavlja problem ukoliko stil generisanog koda odstupa od stila kojim je formatiran originalni kod, što je gotovo uvijek slučaj osim ako je programer od samog početka koristio isti alat za automatsko formatiranje koda. Za alate kao što su alati za refaktorisanje, poželjno je da rezultujući kod bude u potpunosti konzistentan sa originalnim, pa je poželjno koristiti pristupe koji u što većoj mjeri čuvaju formatiranje originalnog koda.

2.3. Komentari

Komentari u kodu su veoma slični formatiranju, u tome što uglavnom nemaju formalnu strukturu i programer ima slobodu da ih koristi kako želi. Kao kod formatiranja, neophodno je obraćati pažnju na povezivanje komentara sa odgovarajućim elementima koda. Međutim, problem povezivanja komentara sa kodom je mnogo ozbiljniji u odnosu na formatiranje. Dok pogrešno formatiranje može izazvati vizuelne nekonzistentnosti, komentari koji se nalaze na pogrešnoj lokaciji mogu u potpunosti izgubiti značenje, ili čak dobiti potpuno drugo značenje u zavisnosti od konteksta.

Određivanje veza između komentara i čvorova stabla se najčešće vrši upotrebom raznih heuristika, tj. jednostavnih pravila koja u praksi uglavnom daju dobre rezultate. Na primjer, može se pretpostaviti da se komentari koji se nalaze samostalno u jednoj liniji odnose na liniju koda nakon sebe, dok se komentari u istoj liniji sa kodom odnose na kod koji im prethodi. Ovo jednostavno pravilo će u većini slučajeva dati dobre rezultate, ali to nije zagarantovano jer programeri imaju potpunu slobodu da koriste komentare kako žele. Za precizne rezultate potrebna je mnogo dublja analiza, uključujući vizuelni izgled koda, položaj komentara u odnosu na kod i druge komentare, pa čak i sadržaj samih komentara i koda koji se nalazi u blizini. Čak i sa naprednom analizom, bez prepoznavanja značenja komentara nije moguće dobiti rezultate koji su u potpunosti ispravni [5].

2.4. Novogenerisani AST čvorovi

Čuvanje originalnog formatiranja koda je veoma bitno kod alata koji proizvode kod namijenjen za dalji razvoj. Međutim, Ovo nije uvijek moguće uraditi. Naime, prilikom refaktorisanja koda nije rijedak slučaj da se kreiraju novi elementi koda koji nisu postojali u originalnom kodu. Na primjer, kod *Extract Method* refaktorisanja, potrebno je kreirati novu funkciju, kopirati dio koda u nju, i zatim kreirati naredbu koja poziva datu funkciju na mjestu gdje se prethodno nalazio dati kod. U ovakvim slučajevima poželjno je kombinovati tehnike čuvanja originalnog formatiranja sa *pretty printing*-om tj. generisanjem novog formatiranja. Na taj način će svi postojeći elementi zadržati originalno formatiranje, a novi čvorovi će imati novo formatiranje. Napredniji oblik ovog pristupa podrazumijeva prepoznavanje stila originalnog formatiranja i primjena datog stila i na nove čvorove kako bi bili konzistentni sa ostatkom koda.

3. IMPLEMENTACIJA

U nastavku će biti predstavljena praktična implementacija koncepata koji su istraženi u radu. Rješenje će biti implementirano kao proširenje alata za razvoj jezika specifičnih za domen TextX i testirano koristeći primjere programa napisanih u jednom novom jeziku, kao i implementacijom jednostavnog alata za refaktorisanje koji se oslanja na dato rješenje. Sve izvršene modifikacije, kao i primjeri koji su korišćeni za testiranje, dostupni su u repozitorijumu https://github.com/ajokic1/pprint-textx.

3.1. TextX

TextX [6] je alat za razvoj jezika specifičnih za domen napisan u Python-u. Dizajniran je za veoma brz i jednostavan razvoj jezika specifičnih za domen. TextX iz jedinstvene definicije gramatike automatski generiše meta-model i parser, koji se zatim mogu koristiti za parsiranje programa i generisanje modela, koji predstavlja stablo apstraktne sintakse programa. Pri tome, TextX funkcioniše kao interpreter, što ga čini veoma dinamičnim i pogodnim za jednostavan razvoj jezika koji su podložni

velikim promjenama, što je veoma značajno u ranim fazama razvoja. TextX je zasnovan na Arpeggio [7] parseru, rekurzivnom silaznom parseru sa vraćanjem i memoizacijom, zasnovanom na PEG gramatikama.

Pri kreiranju novog jezika, TextX prvo parsira gramatiku i na osnovu nje generiše parser i meta-model (apstraktnu reprezentaciju gramatike jezika). Meta-modelu se zatim može proslijediti izvorni kod programa u datom jeziku, pri čemu TextX prvo poziva generisani parser koji kreira stablo parsiranja, a zatim se na osnovu datog stabla konstruiše sami model programa.

3.2. Štampanje

Kako bi se omogućilo pretvaranje modela nazad u izvorni kod, prvo je potrebno obezbijediti da model sadrži sve neophodne elemente koda. Dok stablo parsiranja sadrži sve terminale iz originalnog koda, model sadrži samo one elemente koji su bitni za semantiku koda. Prema tome, ovi elementi se gube prilikom konstruisanja TextX modela, pa je potrebno izvršiti modifikacije u logici za kreiranje modela.

Ovo je postignuto dodavanjem logike koja svaki terminal i neterminal koji se obradi prilikom kreiranja jednog čvora modela čuva u privremenoj promjenljivoj. Pri konačnom kreiranju čvora modela, sadržaj ove promjenljive se čuva kao atribut čvora modela pod nazivom `_tx_pprint_data`.

Nakon dopunjavanja modela na ovaj način, model se može pretvoriti u tekst jednostavnim prolaskom kroz stablo i štampanjem sadržaja pomenutog atributa, ili rekurzivnim pozivanjem funkcije za štampanje nad ostalim čvorovima. Na slici 1. prikazana je osnovna implementacija funkcije za štampanje, koja će kasnije biti proširena

```
def process_node(self, node):
    for n in node._tx_pprint_data:
        n_class = n.__class__.__name__

        if isinstance(n, Terminal):
            self._pprint_terminal(n)
        else:
            self.process_node(n)

def _pprint_terminal(self, node):
    self.append(' ')
    self.append(node)
```

Slika 1. *Osnovna funkcija za štampanje modela*

3.3. Čuvanje formatiranja

Rezultat prethodnog proširenja je stablo koje u većini slučajeva može generisati ispravan kod, ali bez bilo kakvog formatiranja - svi terminali će biti odštampani redom u jednoj liniji. Sada je potrebno obezbijediti da pored odbačenih terminala budu sačuvani i prazni karakteri, tj. formatiranje koda. U ovom slučaju, odbacivanje praznih karaktera se vrši na nivou Arpeggio parsera, prilikom konstruisanja samog stabla parsiranja.

Prema tome, potrebno je izvršiti modifikacije u samom parseru. U podrazumijevanoj konfiguraciji, Arpeggio u potpunosti preskače prazne karaktere prilikom parsiranja. Za čuvanje formatiranja je potrebno obezbijediti da oni ostanu sačuvani i vezani za čvorove stabla. Ovo je moguće uraditi prikupljanjem praznih karaktera u privremenu promjenljivo tokom parsiranja, i njihovim vezivanjem za čvor koji predstavlja naredni terminal. Nakon što su prazni karakteri za svaki terminal sačuvani u vidu atributa čvorova na stablu parsiranja, ovi podaci će biti dostupni u atributu `_tx_pprint_data` iz prethodnog koraka, čime će biti omogućeno štampanje koda sa originalnim formatiranjem.

Međutim, ukoliko se svi prazni karakteri između dva terminala vežu direktno za naredni terminal, ovo može dovesti do neželjenog ponašanja u pojedinim situacijama, jer ovakva veza nije uvijek logički opravdana. Ovo se može uočiti na primjeru na slici 2.

```
function main(test1, test2) {
  x = 20;
  y = 20;
  radius1 = 10;
  radius2 = 15;

  circle x y radius1;
  circle x y radius2;
}
```

```
function main(test1, test2) {
  circle x y radius2;

  circle x y radius1;
  radius2 = 15;
  radius1 = 10;
  y = 20;
  x = 20;
}
```

Slika 2. *Pogrešno čuvanje formatiranja prilikom refaktorisanja*

Nakon zamjene pozicija određenih linija koda, prazna linija koja razdvaja djelove koda ostaje vezana za pogrešan čvor, što dovodi do pogrešnog rezultata. Ovaj problem se može riješiti dodatnom analizom i obradom sačuvanih podataka, tako da se prazne linije tretiraju kao zasebni entiteti i da ne budu vezane direktno za naredni čvor.

3.4. Čuvanje komentara

Komentari se u Arpeggio parseru tretiraju na sličan način kao prazni karakteri. Prema tome, rješenje za čuvanje komentara je relativno slično rješenju za formatiranje - komentari se prikupljaju zajedno sa svojim pozicijama i vezuju se za atribute čvorova u stablu. Međutim, kod komentara je potrebno pokriti neke granične slučajeve kako bi se obezbijedilo čuvanje komentara na odgovarajućim pozicijama u odnosu na kod. Primjer ovakvog slučaja dat je na slici 3. Iz datog primjera može se uočiti da drugi komentar sa tekstom "Second circle" nije vezan za čvor na koji se zapravo odnosi, pa nakon refaktorisanja ostaje na pogrešnoj poziciji. Problem je nastao zato što se svi komentari vezuju za naredni terminal, što u ovom slučaju znači da komentar biva vezan za vitičastu zagradu koja označava završetak funkcije. Ovo se može riješiti uvođenjem pravila koje će komentare koji se nalaze u istoj liniji sa dijelom koda vezati za taj kod. Uz ovu modifikaciju, nakon refakto-risanja komentar ostaje na ispravnoj poziciji.

```
function main(test1, test2) {
  x = 20;
  y = 20;
  radius1 = 10;
  radius2 = 15;

  // Draw two circles with the same center
  circle x y radius1;
  circle x y radius2; // Second circle
}
```

```
function main(test1, test2) {
  // Draw two circles with the same center
  circle x y radius1;
  circle x y radius2;
  x = 20;
  y = 20;
  radius1 = 10;
  radius2 = 15;

   // Second circle
}
```

Slika 3. *Pogrešno čuvanje komentara prilikom refaktorisanja*

3.5. Novogenerisani čvorovi

Kao što je pomenuto, prilikom refaktorisanja može doći i do kreiranja čvorova u stablu koji nisu postojali u originalnom kodu. Za ove čvorove nije moguće sačuvati originalno formatiranje već se ono mora generisati na osnovu određenih pravila. Ovo se može postići dizajniranjem proširive klase za štampanje modela koja će pored štampanja sačuvanog formatiranja omogućiti korisniku da definiše dodatna pravila za štampanje elemenata stabla koji ne posjeduju sačuvano formatiranje. Ovo je implementirano tako da korisnik kreira funkciju za obradu određenog tipa čvora u stablu, a zatim ga dinamički registruje u klasi za štampanje.
Radi demonstracije ove funkcionalnosti, implementiran je jednostavan alat za automatsko vršenje *"Extract method"* transformacije refaktorisanja, koja podrazumijeva izdvajanje dijela koda u zasebnu funkciju. Pri ovoj transformaciji, generiše se nova funkcija, kao i poziv date funkcije, koji predstavljaju novogenerisane čvorove u stablu i formatiraju se korišćenjem definisanih pravila. Rezultat ove transformacije može se vidjeti na slici 4.

4. ZAKLJUČAK

Dok osnovna implementacija transformacije stabla apstraktne sintakse u programski kod nije pretjerano kompleksna, za dobijanje kvalitetnih rezultata neophodno je pokriti veliki broj graničnih slučajeva. Potrebno je vršiti dublju analizu koda i uvoditi pretpostavke o načinu upotrebe nesemantičkih elemenata u većini programa. Rješenja koja su implementirana kao proširenje alata za razvoj jezika poput TextX-a značajno olakšavaju implementaciju funkcionalnosti koje se oslanjaju na ovakve transformacije kod jezika specifičnih za domen, jer će jezici kreirani u ovom alatu već imati podršku za ovakve transformacije. Pri tome, predstavljeno rješenje se može dodatno proširivati dodatnim pravilima za prepoznavanje veza između čvorova stabla i nesemantičkih elemenata koda, čime se mogu dobiti još bolji rezultati.

```
function main(test1, test2) {
  x = 20;
  y = 20;
  radius1 = 10;
  radius2 = 15;

  print_circles(x, y, radius1, radius2);
}

function print_circles (x, y, radius1, radius2) {
  // Draw two circles with the same center
  // but different radiuses
  circle x y radius1;
  circle x y radius2; // Second circle
}
```

Slika 4. *Extract Method refaktorisanje*

4. LITERATURA

[1] Kent Beck, Cynthia Andres, "*Extreme Programming Explained: Embrace Change (2nd Edition)*", Addison-Wesley Professional, 2004.

[2] J. Overbey, R. Johnson, "Generating rewritable abstract syntax trees", *SLE*, 2008.

[3] M. de Jonge, "Pretty-printing for software reengineering", *International Conference on Software Maintenance*, 2002. Proceedings, pp. 550–559, 2002.

[4] M. de Jonge, E. Visser, "An algorithm for layout preservation in refactoring transformations.", *SLE*, 2011.

[5] M. Van De Vanter, "Preserving the Documentary Structure of Source Code in Language-Based Transformation Tools", *SCAM*, pp. 133-143, 2011.

[6] I. Dejanović, R. Vaderna, G. Milosavljević, Ž. Vuković, "TextX: A python tool for domain-specific languages implementation", *Knowledge-Based Systems*, 115:1-4, 2017.

[7] I. Dejanović, G. Milosavljević, R. Vaderna, "Arpeggio: A flexible PEG parser for Python", *Knowledge-Based Systems*, issn:0950-7051, vol.95, p.71–74, DOI:10.1016/j.knosys.2015.12.004, 2016.

Kratka biografija:

Andrej Jokić rođen je u Kotoru 1996. god. Osnovne studije je završio na Elektrotehničkom fakultetu u Podgorici 2019. godine. Master rad na Fakultetu tehničkih nauka iz oblasti Elektrotehnike i računarstva – računarstvo i automatika odbranio je 2021.god.

Kontakt: aj.jokic@gmail.com