Univerzitet u Beogradu

Master rad

Programska realizacija metaprogramiranja u fazi kompilacije u programskom jeziku Skala upotrebom makroa

Jelena Lošić
Matematički fakultet

Beograd 2017

Za buraza

Sadržaj

1 Uvod u metaprogramiranje

Metaprogramiranje je pisanje računarskih programa koji mogu druge programe ili njihove delove koristiti kao svoje podatke. Ova paradigma softverskog razvoja doprinosi mnogim pristupima koji unapređuju programersku produktivnost, što uključuje generisanje koda, analizu programa i pisanje DSL-a (eng. *Domain-specific language*). Ovakav pristup znači da program može biti dizajniran da čita, generiše, analizira ili transformiše druge programe, kao i da menja sebe. U nekim slučajevima, to dopušta programerima da pojednostave svoj kod (zapišu ga konciznije ili čitljivije) i da smanje vreme razvoja programa. Takođe metaprogramiranje može omogućiti da se neki delovi programa koriste u obliku objektnog koda tokom kompilacije i na taj način se ubrzava proces ponovne kompilacije. Programske platforme često prave razliku između metaprogramiranja u vreme kompilacije i onog u vreme izvršavanja programa, u zavisnosti od faze životnog veka programa kada se metaprogrami izvršavaju.

Metaprogamiranje se može koristiti i da se izračunavanja koja bi se inače dešavala u vreme izvršavanja programa vrše tokom kompilacije. Jezik u kom je napisan metaprogram je metajezik. Jezik programa koji koristi metaprograme je objektni jezik. Refleksija, takođe poznata kao introspekcija, označava sposobnost programa da ispituje sopstveno stanje i strukturu (na primer da na osnovu jedne instance klase dobije druge informacije o toj klasi). Programski jezici koji poseduju API za refleksiju (na primer Ruby, Java, Scala itd.) su pogodni jezici za metaprogramiranje.

1.1 Upotreba metaprogramiranja

Ukoliko se radi o velikoj aplikaciji gde veliki broj funkcija uključuje dosta delova koda koji se često ponavljaju (eng. *boilerplate*), može se izdvojiti manji jezik koji će izvršavati te delove koda za programera, a njemu prepustiti izradu ključnih delova.

U mnogim jezicima je prisutno pisanje opširnih iskaza zarad implementacije jednostavnih problema, a to može biti posledica složenosti programskog jezika ili posledica baratanja složenim strukturama podataka. Tada je poželjna upotreba metaprogramiranja da bi se takvi iskazi skratili i smanjila verovatnoća greške u tipovima i druge slične greške.

1.2 Primeri

Osnovni tekstualni makro jezici

Programi koji generišu kod dopuštaju razvoj i upotrebu manjih, domenski specifičnih jezika koji su lakši za pisanje i održavanje od objektnog koda.

C preprocesor - CPP

Tekstualni makroi su makroi koji direktno utiču na tekst programskog jezika bez poznavanja njegove semantike. Jedan od najpoznatijih tekstualnih sistema sa makroima je C preprocesor, a verovatno najpoznatiji primer su makroi koji se definišu pomoću preprocesorske direktive #define. Tekstualna makro ekspanzija je lak način da se vrši osnovno metaprogramiranje u jezicima koji ne poseduju bolje sposobnosti generisanja koda.

```
1  #define SWAP(a, b, type) { type __tmp_c; c = b; b = a; a = c; }
```

SWAP makro omogućava zamenu dve vrednosti datog tipa. Ovu funkciju je najbolje pisati u vidu makroa iz više razloga:

- Poziv funkcije bi potencijalno zahtevao veći utrošak memorije za jednostavnu operaciju.
- Trebalo bi pisati različitu funkciju za svaki tip promenljivih za koje bi se vršila zamena.

Tekstualna supstitucija je korisna, ali takodje ima i ograničeno svojstvo. Problemi na koje se nailazi su:

- Zbog sistema tipova u programskom jeziku C, često su potrebni različiti makroi za različite vrste argumenata. Alternativa je prosleđivanje tipova parametara u vidu argumenata.
- Zbog toga što se vrši tekstualna zamena, C ne može da preimenuje privremene promenljive ako su u konfliktu sa nekim od argumenata koji su prosledjeni. U slučaju $SWAP$ makroa bi došlo do greške ako bi se prosledila promenljiva imenovana sa ___tmp_c.

U programskom jeziku C nailazimo na problem kombinovanja makroa sa izrazima. Posmatrajmo sledeći makro MIN koji vraća manju od dve prosledjene vrednosti.

```
1  #define MIN(x, y) ((x) > (y) ? (y) : (x))
2
```

Veći broj zagrada je nužan zbog prioriteta operatora. Primera radi, pri pozivu MIN(27, b = 32), u slučaju makroa bez zagrada izraz bi se proširio u $27 > b = 32$? $b = 32 : 27$, što bi dovelo do greške u kompajliranju jer bi se prvo povezali 27 > b zbog prioriteta. Postoji i drugi problem, svaka funkcija pozvana kao parametar biće pozvana kad god se na nju naiđe na desnoj strani iz razloga što preprocesor ne poznaje semantiku jezika C i sve što radi je tekstualna zamena. Dakle, ako je dato MIN(do_long_calc(), do_long_calc2()), to će se dalje proširiti u ((do_long_-calc()) > (do_long_calc2()) ? (do_long_calc2()) : (do_long_calc())). Ovo zahteva dodatno vreme, zato što će se bar jedna od metoda pozvati dva puta. Situacija se dodatno pogoršava, ako neko od izračunavanja ima bočne efekte (poput štampanja, menjanja vrednosti globalnih promenljivih), jer će se izvršiti dva puta. Ovaj problem višestrukog poziva može čak dovesti do toga da makro vrati pogrešnu vrednost ako jedna od funkcija vraća različitu vrednost pri svakom pozivu.

Generatori programa

GNU/Linux sistemi podržavaju nekoliko generatora programa, od kojih su najpoznatiji:

- Flex, generator leksičkih analizatora
- Bison, generator parsera
- Gperf, generator heš funkcija

Navedeni programi mogu da generišu izlaz u različitim programskim jezicima (C, Python ...). Razlog zašto su implementirani kao generatori koda, a ne kao funkcije su:

- Ulazi za te funkcije su veoma složeni i teško ih je izraziti u formi koja je valjan odgovarajući kod.
- Ovi programi izračunavaju i generišu mnoge statičke tabele pretrage (*lookuptable*), koje je efikasnije generisati jednom tokom prekompilacije, nego tokom svakog poziva programa.

- Mnogi aspekti funkcionisanja navedenih sistema su prilagodljivi sa proizvoljnim kodom smeštenim na specijalnim pozicijama. Taj kod može koristiti promenljive i funkcije, koje su deo generisane strukture izgrađene od strane generatora koda.

Svaki od gore navedenih alata je fokusiran na izgradnju posebnog tipa programa, *Bison* se koristi za generisanje parsera, *Flex* za leksičke analizatore... Razlikuju se od alata koji su više fokusirani na automatizaciju specifičnih aspekata programiranja. Jedan od primera takvih alata je integracija metoda za pristup bazama podataka u imperativne jezike. Da bi se ovaj postupak učinio lakšim i standardizovanijim, *EmbeddedSQL* je metaprogramski sistem korišćen za lakše pristupanje bazi. Primer jezika sa kojim je izvršena takva integracija je C. Iako su dostupne mnoge biblioteke koje omogućuju pristup relacionim bazama u jeziku *C*, korišćenje generatora koda kao što je *EmbeddedSQL*, čini pristup bazi mnogo prirodnijim, intuitivnijim, i manje podložnim greškama za programera, nego što to čini direktna upotreba biblioteka.

Primer upotrebe *EmbeddedSQL*-a:

```
#include <stdio.h>

int main()

{

    /* Podesavanje konekcije ka bazi */

    EXEC SQL CONNECT TO unix:postgresql://localhost/test USER postgres/
    password;



    /*Sledece promenljive ce se koristiti kao privremeno skladiste
    podataka iz baze.  */

    EXEC SQL BEGIN DECLARE SECTION;

    int my_id;

    VARCHAR my_name[200];

    EXEC SQL END DECLARE SECTION;



    /* Upit koji ce se izvrsiti*/

    EXEC SQL DECLARE test_cursor CURSOR FOR

        SELECT id, name FROM people ORDER BY name;

```

```
31
32
33      /* Izvrsavanje naredbe*/
34
35      EXEC SQL OPEN test_cursor;
36
37
38
39      EXEC SQL WHENEVER NOT FOUND GOTO close_test_cursor;
40
41      while(1) /* prethodna naredba ce se baviti izlaskom iz petlje */
42
43      {
44
45          /* Fetch the next value */
46
47          EXEC SQL FETCH test_cursor INTO :my_id, :my_name;
48
49          printf("Dohvaceni ID ije %d i dohvaceno ime je %s\n", my_id,
        my_name.arr);
50
51      }
52
53
54
55      /* Ciscenje */
56
57      close_test_cursor:
58
59      EXEC SQL CLOSE test_cursor;
60
61      EXEC SQL DISCONNECT;
62
63
64
65      return 0;
66
67  }
```

Metaprogramiranje u programskom jeziku Scheme

Iako generatori koda imaju neko razumevanje objektnog jezika, oni uglavnom nisu potpuni parseri i ne mogu uzeti u obzir objektni jezik, bez prezapisivanja kompletnog kompajlera (termin prezapisivanje u ovom kontekstu označava prevođenje objektnog koda u metajezik). Kako god, ovaj postupak može biti uprošćen ako je reč o jeziku predstavljenom pomoću jednostavne strukture podataka. U programskom jeziku Scheme, sam program je predstavljen kao povezana lista. Scheme programski jezik je

napravljen za obradu listi, što ga čini gotovo idealnim za pisanje programa koji će biti transformisani. Standard jezika Scheme definiše makro jezik, specijalno napravljen da omogući proširivanje jezika. Većina implementacija Scheme-a omogućuju dodatna svojstva za izgradnju programa koji generišu programe [1].

```
;;Definisati SWAP da bude makro
(define-syntax SWAP
   ;;Poziva se syntax-rules metod
   (syntax-rules ()
      ;;Grupa pravila
      (
         ;;Ovo je sablon sa kojim vrsimo poklapanje
         (SWAP a b)
         ;;Ovo je ono u sta zelimo sa ga transformisemo
         (let (
               (c b))
            (set! b a)
            (set! a c)))))

(define first 2)
(define second 9)
(SWAP first second)
(display "first is: ")
(display first)
(newline)
(display "second is: ")
(display second)
(newline)
```

Postoji nekoliko makro sistema za Scheme, ali *syntax-rules* je standard, koji je prikazan u prethodnom primeru. U okviru ovog sistema, *define-syntax* je ključna reč koja se koristi za definiciju makro transformacije. *Syntax-rules* je vrsta transformacije koja kad se primenjuje, unutar zagrada nema makro-specifičnih simbola, sem imena makroa (u primeru iznad nema simbola uopšte). Nakon toga dolazi niz pravila transformisanja. Transformator sintakse će ići od pravila do pravila i pokušati da nađe poklapanje sa šablonom, a kada ga pronađe izvršiće navdenu transformaciju. U ovom slučaju postoji samo jedan šablon, (SWAP a b), gde su a i b promenljive u šablonu koje se ubacuju u jedinice koda u pozivu makroa. Ovde dolazi do poboljšanja u odnosu na makroe iz jezika *C*, prvo zato što su tipovi vezani za same vrednosti, ne za imena promenljivih (neće doći do problema oko tipova promenljivih poput onih iz jezika C). Drugo, neće doći do konflikta čak i da se jedna od promenljivih zvala c, a razlog tome je što su makroi u Scheme-u *hygienic*, što znači da su sve privremene promenljive korišćene od strane makroa automatski preimenovane pre nego što dođe do zamene.

Moguća transformacija makroa za razmenu vrednosti je:

```
(define first 2)
(define second 9)
(let
   (
      (__generated_symbol_1 second))
   (set! second first)
   (set! first __generated_symbol_1))
(display "first is: ")
(display first)
(newline)
(display "second is: ")
(display second)
(newline)
```

Primer kada nije potrebno da privremene promenljive u makrou ne budu automatski preimenovane se javlja kad su te promenljive dostupna kodu koji se transformiše. Jednostavno deklarisanje promenljive neće to omogućiti, jer će makro *system-rules* izvršiti preimenovanje promenljivih. Zbog toga je uveden makro sistem *syntax-case*. *Syntax-case* makroi su teži za pisanje, ali su moćniji jer im je dostupno skoro celo vreme izvršavanja u Scheme-u za izvođenje transformacije.

U sledećem primeru je prikazana osnovna forma *syntax-case* makroa, makro *at-compile-time* će izvršiti datu formu.

```
;;Definicija makroa
(define-syntax at-compile-time
;;x je sintaksni objekat koji treba da se transformise
(lambda (x)
    (syntax-case x ()
        (
;;Patern isti kao syntax-rules patern
            (at-compile-time expression)
;;with-syntax dopusta dinamicku izgradnju sintaksnih objekata
            (with-syntax
                (
;;sintaksni objekat koji gradimo
                    (expression-value
;;nakon racunanja izraza,
;;transfirmisemo ga u sintaksni objekat
                        (datum->syntax-object
;;sintaksni domen
                        (syntax at-compile-time)
;;quote vrednost da bi postala literal
                        (list quote
;;izracunati vrednost za transformaciju
                            (eval
;;konvertovati izraz iz sintaksne u listnu reprezentaciju
                                (syntax-object->datum
```

```
25                                    (syntax expression))
26 ;;sredina u kojoj se racuna
27                                (interaction-environment)
28 )))))
29 ;;vracamo generisanu vrednost kao rezultat
30 (syntax expression-value))))))
31 (define a
32 ;;vrednost promenljive a je 5 u vreme kompilacije
33 (at-compile-time (+ 2 3)))
```

Data operacija će se izvršiti u vreme kompilacije. Preciznije, izvršiće se u vreme ekspanzije makroa, koje nije uvek isto kao i vreme kompilacije u Scheme sistemima. Svaki izraz dostupan u vreme kompilacije u Scheme sistemu, biće dostupan za korišćenje u ovom izrazu.

Naredbom *syntax-case*, definiše se transformišuća funkcija, u okviru koje se upotrebljava lambda račun. Argument x je izraz koji se transformiše. $With - syntax$ definiše dodatne sintaksne elemente koji se mogu koristiti u izrazu koji se transformiše. *Syntax* prihvata sintaksne elemente, kombinuje ih, prateći ista pravila kao transformator u *syntax-rules*. U programu iz prethodnog primera izvršavaju se sledeći koraci:

1. Izraz *expression* u *at-compile-time* se prihvata.
2. U najugnježdenijem delu transformacije, *expression* se konvertuje u listu i tretira se kao standardan scheme kod.
3. Rezultat se potom kombinuje simbolom *quote* u listu, te će ga Scheme tretirati kao literal kada postane kod.
4. Ovaj podatak se konvertuje u sintaksni objekat.
5. Tom sintaksnom objektu je dato ime *expression-value* kako bi se prikazao na izlazu.
6. Transformator (*syntax expression-value*) govori da je *expression-value* sveukupan izlaz iz ovog makroa.

1.3 Refleksija i metaprogramiranje

Termin metaprogramiranje u ovom radu se odnosi na programe koji generišu ili modifikuju druge programe, dok termin refleksije (introspekcije) referiše na reprezentaciju aspekata sistema u njemu samom. Metaprogramiranje i refleksija su usko

povezani; metaprogrami mogu koristiti reflektovane informacije o svom kontekstu u stvaranju programa. Brajan Smit[4] je 1984. zasnovao tekuću tradiciju istraživanja refleksije u programskim jezicima. On se fokusirao na to da Lisp interpreter bude sposoban da reflektuje sopstvenu operacionu semantiku. Ovaj stil refleksije se naziva dinamička refleksija i omogućava pristup meta informacijama tipova. Ovde će biti više reči o refleksiji koja se vrši u vreme kompilacije.

Kvazinavodi u programskim jezicima

Kvazinavode je osmislio Kvini[1] u svojoj knjizi *Matematička logika* koja je objavljena 1940. godine [3]. Dok obični navodi podrazumevaju spominjanje fraze, pre nego njenu upotrebu, kvazinavodi dopuštaju da izrazi pod navodnicima sadrže druge promenljive koje zamenjuju neke druge izraze, kao što matematički izrazi mogu sadržati promenljive koje imaju određene vrednosti. Paradigmatska instanca kvazinavoda u programskim jezicima se sreće u Lisp i Scheme familijama jezika. U Lisp familiji, programski kod se predstavlja uniformno, korišćenjem listi koje sadrže atomične podatke, kao što su simboli, niske, brojevi ili liste. U Lispu se ove strukture nazivaju S-izrazi. Zato što su S-izrazi obični podaci, ima smisla stavljati ih pod navode, na taj način stvarajući strukturu lakšu za obradu. Većina Lisp jezika ima sistem za kvazinavode, u kome se specijalno označeni podizrazi izračunavaju, a rezultat se zamenjuje pod navodima. Za razliku od Kvinijevih kvazinavoda, Lisp familija jezika dopušta umetanje proizvoljnih izraza u kvazinavode [6].

Primer jezika, koji koriste kvazinavode radi implementacije proširenja jezika je OCaml. Kvazinavodi u Camlp4 sistemu se sastoje od proizvoljnih niski koje se transformišu *quotation expander* alatom u nisku koja predstavlja konkretnu sintaksu ili apstraktno sintaksno drvo. Ovi navodi podržavaju i antinavode, koji pozivaju parser da iščita OCaml izraz ili obrazac unutar navoda. Šablonski (eng. *template*) kvazinavodi u Haskell-u rade na sličnim principima. Oba jezika u potpunosti proširuju navode u vreme kompilacije, i oba proveravaju da li je generisani kod korektno tipiziran [2].

Kvazinavodi u programskom jeziku Skala su veoma slični Lisp kvazinavodima. Njihova sintaksa je slična sintaksi niski, što je posledica njihove implementacije koja koristi Skaline interpolatore za niske i oni se u vreme kompilacije proširuju u drvoidne strukture. Kvazinavodi su inicijalno služili kao tehnika za implementiranje makroa u Skali, ali su se pokazali korisnima za generisanje koda u vreme izvršavanja kao i za generisanje programskog teksta. Skalini makroi vode poreklo od makroa u Lispu, u

[1] Willard van Orman Quine 1908-2000

pogledu toga što implementiraju transformacije iz jednog parsiranog drveta u drugo. Kao i Scala, C# je objektno orijentisani jezik u kome se koriste kvazinavodi. U C# navodi se mogu primeniti na anonimne funkcije, anotirajući ih sa *Expression* tipom, što dovodi do generisanja AST (eng. Abstract Syntax Tree) koje odgovara funkciji, umesto funkcije same. Ipak, ovo svojstvo se ne može smatrati pravim kvazinavodima, jer nije podržan mehanizam za izbegavanje navoda i umetanje poddrveta koje je generisano na nekom drugom mestu u kodu.

Statička refleksija

Za razliku od dinamičke refleksije gde se meta informacijama tipova i članovima može pristupiti preko tipske klase, u slučaju statičke refleksije koja se spominjr u ovom radu, isti efekat se postiže introspekcijom drveta izraza. Raketa (eng. *Racket*) je među jezicima koji su najnapredniji po pitanju statičke refleksije. Raketina svojstva "jezici i biblioteke"(eng. *languages and libraries*) omogućuju definisanje potpuno novih jezika u Raketi. Svaki modul specificira jezik na kome je napisan, a jezici su definisani bibliotekama koje kontrolišu parsiranje i ekspanziju modula u osnovni (eng. *core*) jezik. Ove biblioteke omogućuju upotrebu proizvoljnog koda napisanog u Raketi tokom ekspanzije i imaju pristup okruženju koje poseduje bogatu metadatu o sistemu. I Template Haskell i Skalin makro sistem podržavaju pretraživanje globalnog stanja kompajlera tokom izvršenja metaprograma, koristeći refleksiju za generisanje koda i metaprogramiranje.

Popularni objektno orijentisani jezici poput Jave i C# imaju slabu podršku za statičku refleksiju, premda postoje sistemi statičkog reflektivnog metaprogramiranja za njih u kojima se primenjuje pristup refleksije zasnovan na šablonima (klase mogu biti definisane rekurzijom po strukturi drugih klasa). U ovim sistemima je akcenat stavljen na bezbednost metaprograma. Donekle je i nejasno u kojoj meri bi ove alate trebalo klasifikovati u refleksiju, jer jezik korišćen za reprezentaciju i procesiranje definicija nije programski jezik sam, već domenski specifičan jezik.

2 Pregled programskog jezika Skala

2.1 Uvod

Skala je namenjena za konstrukciju komponenti i komponentnih sistema. Pravi komponentni sistemi su do danas ostali nedostižan cilj u softverskoj industriji. Softver bi trebalo sastaviti od biblioteka sa unapred napisanim komponentama, baš kao što je i hardver sastavljen od unapred izrađenih čipova. U realnosti, veliki delovi aplikativnih programa su napisani "od nule", pa je proizvodnja softvera i dalje zanat, a ne industrija.

Komponente u ovom smislu su jednostavne softverske jedinice koje se na neki način koriste od strane aplikacije. Pojavljuju se u više oblika: mogu biti moduli, klase, biblioteke, radna okruženja, procesi ili veb servisi. Mogu se povezivati sa drugim komponentama putem agregacije, parametrizacije, nasleđivanja, prosleđivanja poruka.

Sporiji napredak komponentnog softvera je posledica mana programskih jezika korišćenih za definisanje i integrisanje komponenti. Većina postojećih jezika nude ograničenu podršku za apstrakciju komponenti i njihovo sastavljanje. Ograničenost podrške se najviše odnosi na statički tipizirane jezike, kao što su Java i C# u kojima je napisan dobar deo današnjeg softvera.

Razvoj Skale je započet 2001. godine u laboratorijama EPFL-a, a postaje javno dostupna od januara 2004. godine. Rad na Skali počiva na istraživačkom naporu da se napravi jezik koji će biti bolja podrška komponentnom softveru[5]. Postoje dve hipoteze na kojima se bazira taj rad. Prva je da programski jezik za komponentni softver mora biti skalabilan u smislu da isti koncepti važe, kako za male, tako i za velike delove. Druga je da se skalabilna podrška za komponente obezbeđuje programskim jezikom koji ujedinjuje objektno orijentisano i funkcionalno programiranje.

2.2 Skalabilan jezik

Skala je zamišljena kao jezik koji će biti sposoban da raste (eng. *scale*) kako budu rasle potrebe njegovih korisnika, te otud i potiče naziv ovog programskog jezika.

Skala se može primeniti na širok opseg programerskih zadataka, od pisanja malih skripti do izgradnje velikih sistema. Nije teško početi raditi sa Skalom jer se izvršava na Java platformi i moguće je direktno koristiti Java biblioteke.

Tehnički, Skala je mešavina objektno-orijentisanih i funkcionalnih programskih koncepata u statički tipiziranom jeziku (provera tipova se vrši u vreme kompilacije). Njena objektno-orijentisana priroda se ogleda u tome što je svaka vrednost objekat i svaka operacija poziv metoda. Na primer, kada napišemo $1 + 2$, zapravo pozivamo metod + definisan u klasi Int. Skala je naprednija od većine drugih jezika u kombinovanju objekata - primer su crte (eng. *trait*) koje su poput interfejsa u Javi. Objekti mogu nastati kombinovanjem crta, što znači da se uzima definicija klase na koju se dodaju željene razlike iz raznih crta.

Sa druge strane, funkcionalna priroda se ogleda u tome što je funkcija vrednost, sa istim statusom, kao što ga imaju primera radi ceo broj ili niska. Funkcije se mogu proslediti kao argumenti drugim funkcijama, biti povratna vrednost i mogu se čuvati u promenljivim. Funkcija se može definisati unutar druge funkcije, slično definisanju celobrojne vrednosti. Funkcije višeg reda doprinose Skalinoj skalabilnosti, jer omogućava pisanje veoma konciznog koda. Još jedna ideja funkcionalnog programiranja, takođe realizovana u Skali je ta da operacije u programu preslikavaju vrednosti u neke druge vrednosti, umesto da ih menjaju. Nepromenljive strukture podataka omogućavaju sprovođenje navedenog principa. Skala poseduje nepromenljive liste, n-torke, mape i skupove. Drugi način iskazivanja ove ideje je da metode nemaju sporednih efekata.

Ova dva stila programiranja imaju komplementarne osobine kada se radi o skalabilnosti. Funkcionalne konstrukcije u Skali omogućavaju brzo pravljenje prototipova od jednostavnih delova koda, dok njene objektne konstrukcije olakšavaju struktuiranje većih sistema i njihovu prilagodljivost novim zahtevima.

2.3 Statička tipiziranost

Statički sistem tipova klasifikuje promenljive i izraze prema vrstama vrednosti koje čuvaju i nad kojima vrše računske operacije. Skala važi za jezik sa veoma naprednim statičkim sistemom tipova. Počev od sistema ugnježdenih tipova klasa koji je nalik onom iz Jave, Skala dopušta da se tipovi parametrizuju sa generičkim tipovima, da se kombinuju tipovi korišćenjem preseka i da se skrivaju detalji o tipovima korišćenjem apstraktnih tipova.

Konkretno, sistem tipova podržava:

1. generičke klase
2. razne anotacije
3. gornje i donje granice za tipove (u smislu izvođenja)
4. unutrašnje klase i apstraktne tipove kao članove objekata
5. složene tipove
6. reference na same tipove (referenca na samog sebe)
7. polimorfne metode

Mehanizam izvođenja tipova (eng. type inference) se brine da korisnik nije u obavezi da navodi tipove eksplicitno.

Navedena svojstva zajedno čine moćnu bazu za bezbednu ponovnu upotrebu programskih apstrakcija i bezbedno proširenje softvera.

2.4 Apstraktni tipovi

Klase u Skali su parametrizovane vrednostima (parametri konstruktora) i tipovima (ako su klase generičke).

Sledeći primer prikazuje klasu $Buffer$ u okviru koje se nalazi definicija apstraktnog tipa.

```
1 abstract class Buffer {
2     type T
3     val element: T
4 }
```

Apstraktni tipovi su tipovi čiji identitet nije precizno određen. U primeru klase $Buffer$ zna se samo da svaki objekat klase $Buffer$ ima član čiji je tip T. Međutim, definicija klase ne otkriva kojem konkretnom tipu T odgovara.

Definicije tipova se mogu preinačiti (eng. *override*) u podklasama. Time se otkriva više informacija o apstraktnom tipu jer se za njega vezuju granice tipa (koje opisuju mogućnosti realizacije apstraktnog tipa). U sledećem primeru je navedena klasa $SeqBuffer$ koja dopušta čuvanje isključivo nizova u baferu, time što zahteva da tip T mora biti podtip novog apstraktnog tipa U.

```
abstract class SeqBuffer extends Buffer {
    type U
    type T <: Seq[U]

    def length = element.length
}
```

Klase sa članovima apstraktnog tipa se često koriste u kombinaciji sa instanciranjem anonimnih klasa.

2.5 Klase

Klase u Skali su statički obrasci (eng. templates) koji se mogu instacirati u mnoge objekte tokom izvršavanja programa. U sledećem primeru je definisana klasa *Point*:

```
class Point(xc: Int, yc: Int)
{
  var x: Int = xc
  var y: Int = yc

  def move(dx: Int, dy: Int)
  {
    x += dx
    y += dy
  }
}
```

Klase u Skali su parametrizovane argumentima konstruktora. U klasi *Point* definisana su dva argumenta konstruktora *xc* i *yc*, koji su vidljivi u celom telu klase. Prilikom instanciranja klase koristi se ključna reč *new*.

2.6 *Case* klase

Case klase su regularne klase koje eksportuju parametre konstruktora i time omogućuju rekurzivni mehanizam dekompozicije putem prepoznavanja šablona (eng. *pattern matching*).

U narednom primeru se navodi hijerarhija klasa koja se sastoji od apstraktne super klase *Term* i tri konkretne case klase *Var*, *Fun*, *App*.

```
abstract class Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
```

```
case class App(f: Term, v: Term) extends Term
```

Ova hijerarhija se može koristiti za predstavljanje izraza netipiziranog lambda računa. Da bi se olakšalo pravljenje instanci *case* klasa, Skala ne zahteva da se koristi ključna reč *new*. Dovoljno je da se koristi ime klase kao funkcija, što oslikava primer:

```
Fun("x", Fun("y", App(Var("x")), Var("y")))
```

Parametri konstruktora *case* klasa su javne vrednosti i može im se direktno pristupiti. Case klase se zbog toga koriste prilikom dekompozicije struktura podataka pomoću prepoznavanja šablona. U primeru ispod izraz tipa *Term* se lako dekomponuje korišćenjem *case* klasa i prepoznavanja šablona:

```
def print(term: Term): Unit = term match {
  case Var(n) =>
    Console.print(n)
  case Fun(x, b) =>
    Console.print("^" + x + ".")
    print(b)
  case App(f, v) =>
    Console.print("(")
    print(f)
    Console.print(",")
    print(v)
    Console.print(")")
}
```

2.7 Apstrakcija nad nizovima

Skala nudi jednostavnu notaciju za izražavanje apstrakcije nad nizom (eng. *sequence comprehensions*). Apstrakcija ima oblik **for** (*enums*) **yield** *e*, gde *enums* referiše na listu enumeratora razdvojenih tačka-zarezom. Enumerator je ili generator koji uvodi nove promenljive ili je filter. Apstrakcija izračunava telo *e* za svako vezivanje generisano enumeratorom *enum* i vraća niz dobijenih vrednosti.

Primer apstrakcije:

```
def even(from: Int, to: Int) =
{
  for(val i <- List.range(from, to); i % 2 == 0) yield i
}
```

Apstrakcije nisu ograničene samo na nizove (*List* klasa je izvedena iz klase *Seq*), one su podržane za sve strukture podataka koje podržavaju operacije **filter**, **map** i **flatMap**.

2.8 Generičke klase

Poput Jave, Skala ima ugrađenu podršku za klase parametrizovane tipovima. Takve generičke klase su naročito pogodne za razvoj klasa za kolekcije. Evo primera koji to demonstrira:

```
class Stack[T] {
  val elems: List[T] = Nil
  def push(x: T): Unit = elems = x ::  elems
  def pop: Unit = elems = elems.tail()
}
```

Klasa **Stack** modeluje promenljive stekove proizvoljnih elemenata tipa T. Upotreba parametara tipova dopušta da se jedino elementi tipa T mogu smeštati na stek. Generički tipovi su invarijantni u odnosu na podtipove, što znači da ako postoji $Stack[Char]$, on se ne može koristiti kao celobrojni stek tipa $Stack[Int]$.

2.9 Funkcije višeg reda

Skala dopušta definisanje funkcija višeg reda, tj. funkcija koje uzimaju druge funkcije kao parametre ili im je rezultat funkcija. Primer takve funkcije je funkcija *apply* koja uzima kao argument funkciju f i primenjuje je na argument v:

```
def apply(f: Int => String, v: Int) = f(v)
```

2.10 Impliciti

Delovi šablona i parametri označeni modifikatorom **implicit** mogu biti prosleđeni implicitnim parametrima i mogu biti korišćeni kao implicitne konverzije koje se nazivaju pogledi. Implicitni modifikator nije dozvoljen u članovima tipova, kao ni u objektima najvišeg nivoa (misli se na prvi nivo ugnježdavanja unutar paketa)[7].

2.10.1 Implicitni parametri

Metod sa implicitnim parametrima se može primenjivati na argumente kao što to čine i standardne metode. U ovom slučaju labela **implicit** nema nikakvog uticaja. Međutim, ako takvom metodu nedostaju argumenti koji odgovaraju implicitnim parametrima, ti argumenti će automatski biti obezbeđeni. Argumenti kojima je dopušteno da budu prosleđeni implicitnim parametrima spadaju u dve kategorije:

1. Svi identifikatori x kojima se može pristupiti na mestu poziva metoda bez prefiksa i koji označavaju implicitnu definiciju ili implicitni parametar.

2. Svi članovi pratećih modula vezanih za tip parametra koji su označeni sa **implicit**.

U sledećem primeru definiše se metod *sum* koji računa sumu elemenata liste, koristeći operacije *add* i *unit* iz klase *Monoid*.

```
1  abstract class SemiGroup[A] {
2    def add(x: A, y: A): A
3  }
4
5  abstract class Monoid[A] extends SemiGroup[A]{
6    def unit: A
7  }
8
9  object ImplicitTest extends Applicaiton {
10   implicit object StringMonoid extends Monoid[String]{
11     def add(x: String, y: String) = x.concat(y)
12     def unit = ""
13   }
14
15   implicit object IntMonoid extends Monoid[Int]{
16     def add(x: Int, y: Int) = x + y
17     def unit = 0
18   }
19
20   def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
21     if(xs.isEmpty()) m.unit()
22     else m.add(xs.head, sum(xs.tail))
23
24   sum(List(1,2,3))
25   sum(List("a", "b", "c"))
26 }
```

2.10.2 Pogledi

Implicitni parametri i metode takođe mogu definisati implicitne konverzije koje se nazivaju pogledi. Pogled iz tipa S u tip T je definisan implicitnom vrednošću koju ima funkcija tipa S=>T ili metodom koja se može konvertovati u vrednost datog tipa.

Pogledi se primenjuju u sledeće tri situacije:

1. Ako je izraz *e* tipa T, i T se ne poklapa se očekivanim tipom izraza *pt*. U ovom slučaju traži se implicit v koji je primenljiv na e i čiji se tip rezultata slaže sa *pt*. Pretraga se nastavlja kao i u slučaju implicitnih parametara, gde je oblast važenja implicita ona kojoj pripada T => pt. Ako se takav pogled pronađe, izraz se konvertuje u v(e).

2. U selekciji e.m gde je e tipa T, a gde selektor m ne označava član od T kome se može pristupiti. U ovom slučaju, traži se pogled v koji se može primeniti na e i njegov rezultat sadrži član m. Pretraga se obavlja kao i kod implicitnih parametara, gde je oblast važenja implicita ona kojoj pripada T. Ako se takav pogled pronađe, selekcija e.m se konvertuje u v(e).m.

3. U selekciji e.m(args) gde je e tipa T, ako selektor m označava neki član(ili članove) od T, ali nijedan od tih članova nije primenljiv na argumente *args*. U ovom slučaju, traži se pogled v koji je primenljiv na e i čiji rezultat sadrži metod m koji je primenljiv na argumente *args*. Pretraga se odvija kao i kod implicitnih parametara, gde oblast važenja implicita ona kojoj pripada T. Ako se takav pogled pronađe, selekcija e.m se konvertuje u v(e).m(args).

2.11 Izvođenje tipova

Skala poseduje ugrađeni mehanizam za izvođenje tipova koji omogućava programeru da izostavi određene anotacije tipova. Na primer, često nije neophodno navesti tip promenljive, jer kompajler može da izvede tip iz izraza kojim se inicijalizuje promenljiva. Takođe povratni tipovi metoda nekada mogu biti izostavljeni, jer odgovaraju tipu tela metoda koji kompajler izvodi.

Primer:

```
object InferenceTest extends Application {
  val x = 1 + 3 + 5 // x je tipa int
  val y = x.toString() // y je tipa String
  def succ(x: Int) = x + 1 // tip povratne vrednosti je Int
}
```

Za rekurzivne metode, kompajler nije sposoban da izvede tip povratne vrednosti. Primer programa koji će vratiti grešku od strane kompajlera:

```
object InferenceTest2 extends Application{
  def fac(n: Int) = if(n == 0) 1 else n*fac(n-1)
}
```

2.12 Donje granice tipa

Gornje granice tipa ograničavaju tip na podtip drugog tipa. Donje granice tipa deklarišu tip kao supertip nekog drugog tipa. Term T :> A iskazuje da tip parametra T referiše na supertip tipa A.

```
case class ListNode[T](h: T, t: ListNode[T]){
  def head: T = h
  def tail: ListNode[T] = t
  def prepend(elem: T): ListNode[T] =
    ListNode(elem, this)
}
```

Prethodni primer definiše povezanu listu sa operacijom dodavanja elementa na početak liste. Kako je ovaj tip invarijantan u odnosu na tip parametra T, npr. tip ListNode[String] nije podtip tipa List[Object], neophodna je oznaka za varijaciju (u sledećem primeru to je znak + koji je oznaka za kovarijaciju) kako bi se ta nemogućnost otklonila.

```
case class ListNode[+T] (h: T, t: ListNode[T]) { ... }
```

Uvođenjem oznake za varijaciju, program se neće prevoditi, jer je kovarijantna anotacija moguća jedino ako je promenljiva za tip korišćena na kovarijantnim pozicijama. Pošto se tipska promenljiva T pojavljuje kao parametar za tip u metodu *prepend*, ovo pravilo je prekršeno. Uz pomoć oznaka za donju granicu tipa, može se implementirati *prepend* metod gde se T pojavljuje samo na kovarijantnim pozicijama. Primer odgovarajućeg koda.

```
case class ListNode[+T](h: T, t: ListNode[T]){
  def head: T = h
  def tail: ListNode[T] = t
```

```
4   def prepend[U >: T](elem: U): ListNode[U] =
5     ListNode(elem, this)
6 }
```

2.13 Kompozicija klasa uz pomoć primesa

Nasuprot jezicima koji podržavaju jednostruko nasleđivanje (eng. single inheritence), Skala omogućava da se u definiciju nove klase doda razlika u odnosu na super klasu. Ovakav način građenja klasa se naziva kompozicija klasa uz pomoć primesa (eng. mixin class composition). Ako se posmatra apstrakcija za iteratore:

```
1 abstract class AbsIterator {
2   type T
3   def hasNext: Boolean
4   def next: T
5 }
```

Primećuje se da se pri definisanju klase koja se može koristiti kao primesa koristi ključna reč **trait**. Primer klase sa primesama u odnosu na klasu *AbsIterator*:

```
1 trait RichIterator extends AbsIterator{
2   def foreach(f: T => Unit) : Unit{
3     while(hasNext) f(next)
4   }
5 }
```

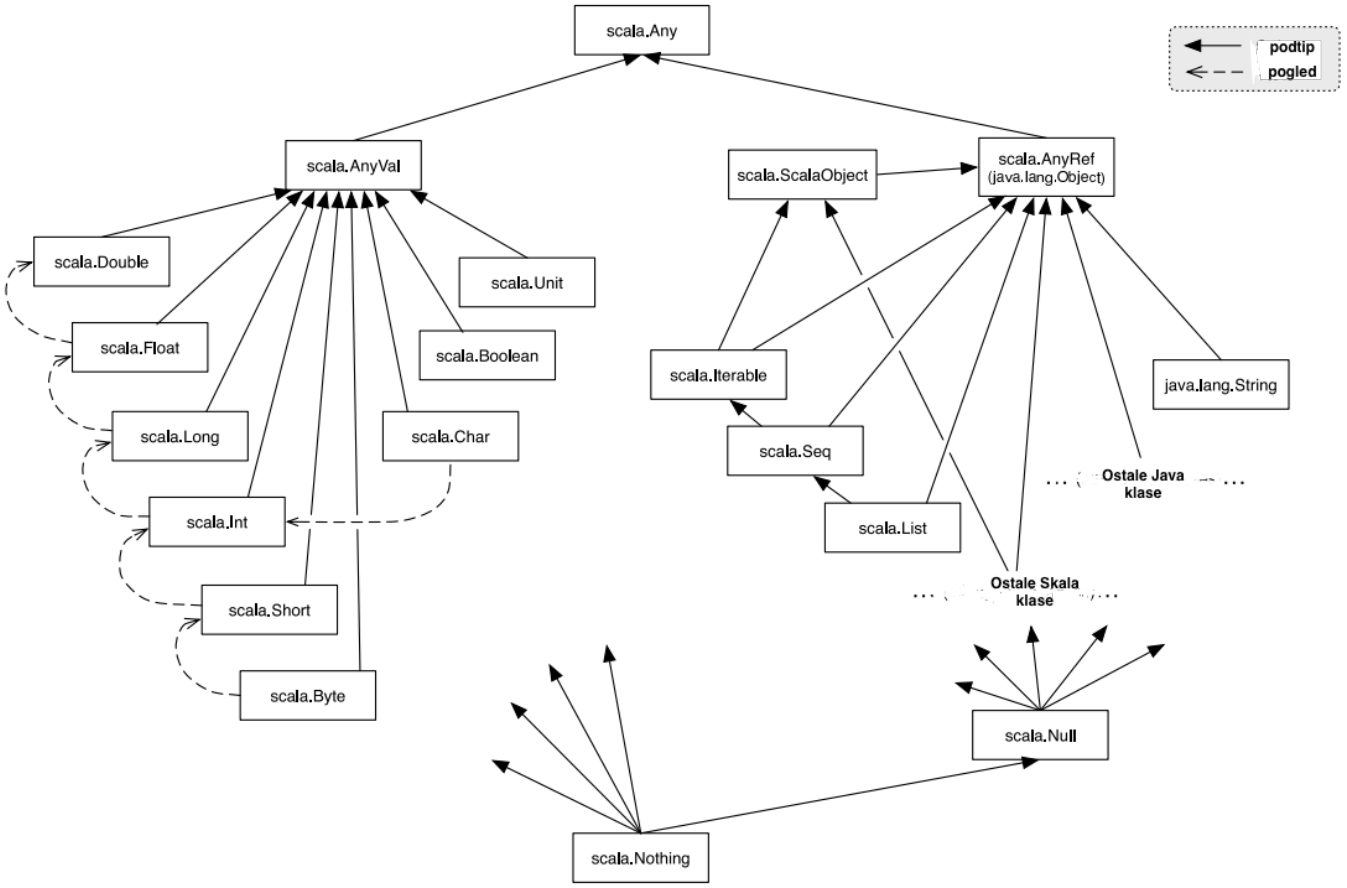
Primer konkretne iteratorske klase:

```
1 class StringIterator(s: String) extends AbsIterator{
2   type T = Char
3   private var i = 0
4   def hasNext = i < s.length()
5   def next = { val ch = s.charAt(i); i = i+1; ch}
6 }
```

Način iskorišćavanja funkcionalnosti *RichIterator* i *StringIterator* klase je kompozicija klasa sa primesama pomoću ključne reči **with**. Prvi roditelj se naziva superklasa klase *Iter*, dok se drugi i svaki naredni naziva primesa.

Primer:

```
1 class Iter extends StringIterator(s: String) with RichIterator
```



Slika 2.1: Dijagram hijerarhije klasa u Skali.

2.14 Unifikovani tipovi

Jedna od značajnijih odlika Skale je njen unifikovani sistem tipova (hijerarhija tipova je prikazana na sledećem dijagramu). Za razliku od Jave, sve vrednosti u Skali su objekti (uključujući numeričke vrednosti i funkcije). Kako je Skala zasnovana na klasama, svaka vrednost je instanca neke klase.

Superklasa svih klasa je *scala.Any* koja ima dve direktne podklase *scala.AnyVal* i *scala.AnyRef* koje predstavljaju klase za vrednosti i klase za reference. Klase za vrednosti su predefinisane, one odgovaraju primitivnim tipovima iz Jave. Svaka korisnički definisana klasa nasleđuje crtu *scala.ScalaObject*.

3 Scala.reflect

Scala.reflect je radni okvir (eng. framework) ili biblioteka za metaprogramiranje u Skali. Biblioteka za metaprogramiranje je kolekcija struktura podataka i operacija koje omogućuju pisanje metaprograma. Najznačajnija upotreba *Scala.reflect*-a se ogleda u makroima koji omogućuju pisanje metaprograma koji se izvršavaju tokom kompilacije. Makroi koriste bogat i stabilan API i potom distribuiraju metaprograme koje pišu u regularne Skaline biblioteke.

3.1 Okruženja

U *scala.reflect*-u okruženja su reprezentovana pomoću klase *Universe* (u daljem nastavku rada, ova klasa će se spominjati pod imenom *univerzum*). Za svako podržano okruženje, postoji specijalna podklasa od *scala.reflect.api.Universe* koja obezbeđuje implementaciju apstraktnog jezičkog modela za to okruženje. Navedena klasa je veoma opširna, u momentu pisanja nasleđuje dvadeset crta, a nad svakom od tih crta zbirno postoji nekoliko stotina javnih API-ja.

```
abstract class Universe extends Symbols
with Types
with FlagSets
with Scopes
with Names
with Trees
with Constants
with Annotations
with Positions
with Exprs
with TypeTags
with ImplicitTags
with StandardDefinitions
with StandardNames
with StandardLiftables
with Mirrors
with Printers
with Liftables
with Quasiquotes
with Internals
```

Dok univerzum proizvodi infrastrukturu za jezički model i implementira operacije koje rade sa elementima modela, on ne zna kako da popuni tabelu simbola koja odgovara određenom okruženju. Taj posao je prosleđen entitetu *Mirror*.

```
abstract class Mirror {
  val universe: Universe
  val RootPackage: universe.ModuleSymbol
  ...
}
```

Kao što se može videti iz gore navedene definicije, ogledalo (eng. *mirror*) ima vezu ka odgovarajućem univerzumu i obezbeđuje simbol koji odgovara korenskom paketu Skalinih programa. Odatle, koristeći potpise simbola, može se stići do svih polaznih definicija i njihovih članova.

Svaki univerzum je propraćen sa jednim ili više ogledala. Tu je uvek *rootMirror* koji reflektuje fundamentalne definicije iz standardne biblioteke. Dodatna ogledala su neophodna, ako okruženje može da definiše podokruženja koja odgovaraju različitim putanjama do klase. (eng. *classpath*)

3.2 Skalin kompajler

U slučaju metaprograma koje izvršava Skalin kompajler, umesto programera koji instancira univerzum i pokreće svoje metaprograme u tom univerzumu, on registruje metaprograme kako bi se izvršili u kompajleru. Skalin kompajler pruža tri vrste ekstenzionih tačaka: 1) kompajlerski priključci koji mogu da registruju nestandardne faze da se izvršavaju nakon uobičajenih, 2) tipski priključci koji mogu da registruju povratne pozive pokrenute u specifičnim tačkama unutrašnje implementacije faze provere tipova, 3) makroi koji se izvršavaju kada *typechecker* (u daljem tekstu *proveravač tipova*) naiđe na specijalno definisane metode, tipove, anotacije i dr. U vreme pisanja *scala.reflect*-a, unutar njega se mogu pisati samo makroi.

Skalin kompajler, tj. *scala.tools.nsc.Global* je jedan od univerzuma *scala.reflect*-a. To je očekivano, jer je *scala.reflect* nastao proširivanjem i profinjavanjem funkcionalnosti unutrašnjeg mehanizma kompajlera. Kako su jezički modeli *scala.reflect*-a i unutrašnjeg mehanizma Skalinog kompajlera veoma slični, izvršavanje *scala.reflect* metaprograma u Skalinom kompajleru je trivijalno. Sve dok su metaprogrami u skladu sa funkcionisanjem kompajlera, dok koriste korektne eliminacije sintaksnih

ulepšavanja i drže tabelu simbola sinhronizovanu sa generisanim kodom, nema problema u izvršavanju. Poznavanje načina rada kompajlera je neophodno radi efikasne upotrebe *scala.reflect*-a.

3.3 Arhitektura Scala.reflect-a

Osnovna ideja dizajna *Scala.reflect*-a je da metaprogramima učini dosupnim unutrašnje mehanizme kompajlera, koji imaju bogat jezički model i skup operacija koje ga prate. Rad na ovoj biblioteci je bio veoma zahtevan, jer je kompajler premašivao 100,000 linija koda pre njenog nastanka.

Global je klasa koja implementira Skalin kompajler - *scalac*. Ona nasleđuje *SymbolTable* klasu. *SymbolTable* je izgrađen upotrebom primesa, konkretno to su crte *Trees*, *Symbols*, *Names*.

Scala.reflect.api.Universe klasa je premosnica (eng. gateway) ka *Scala.reflect* API-ju, koja je takođe izgrađena upotrebom primesa.

U narednom primeru definsana je crta $Immutable[T]$ i makro $Materialize[T]$ koji generiše vrednosti tipa $Immutable[T]$ ako je tip argumenta nepromenjiv, inače program koji koristi dati makro prekida sa greškom. $Materialize[T]$ makro je u suštini adapter za metaprogram $Macros.Impl[T]$ koji će se izvršavati unutar kompajlera (ovde se misli na okruženje koje predstavlja kompajler, a makro zahteva referencu na okruženje u kom se izvršava) kada god se provera tipova susretne sa $Materialize[T]$

```
import scala.language.macros
import scala.reflect.macros.blackbox.Context

trait Immutable[T]
object Immutable {
  implicit def materialize[T]: Immutable[T] = macro Macros.impl[T]
}
object Macros{
  def impl[T](c: Context)(implicit tag: c.WeakTypeTag[T]) = {
    import c.universe._

    def isImmutable(t: Type): Boolean = {
      ...
    }
    val t = tag.tpe
    if(isImmutable(t)) q"null"
    else
      c.abort(c.enclosingPosition, t + " is not immutable")
  }
}
```

Kako makroi spadaju u eksperimenalne mogućnosti jezika, to zahteva i uključivanje dodatne biblioteke. Metaprogram *Impl* uzima kontekst kompajlera koji između ostalog uključuje i *Universe*. Što znači da uključivanje *c.Universe._* donosi ceo *Scala.reflect* API u područje rada (eng. scope), obezbeđujući definicije kao što su *Type*, *ClassSymbol*, *TypeSymbol*, itd.

Pored konteksta, *Impl* prihvata tag tip koji uključuje reprezentaciju argumenta za tip. Ako provera nepromenljivosti bude uspešna, makro će vratiti trivijalnu instancu tipa Immutable. U slučaju neuspešne provere, ekspanzija makroa se takođe završava neuspehom pozivom *c.abort* što će dovesti do greške u kompilaciji. Ova greška će biti pozicionirana na mestu poziva makroa, određenog sa *c.enclosingPosition* i time detaljnije informisati programera. Sposobnost vraćanja domenski specifičnih grešaka se pokazala kao jedna od bitnijih osobina makroa.

Primer korišćenja makroa **materialize**: Neka je dat algoritam **compute** koji za argument ima ulaznu konfiguraciju i koji se izvršava određeno vreme i pritom startuje nove niti koje se izvršavaju paralelno sa glavnim programom. Željeno ponašanje je da nijedna od tih niti ne izmeni ulaznu konfiguraciju. Da bi se to obezbedilo, zahteva se od pozivaoca metoda da pruži dokaz da je konfiguracija nepromenljiva.

```
1 def compute[C <: Config](c: C)(implicit ev: Immutable[C]) = {...}
```

Zahvaljujući Skalinim implicitima, kad god programer ne obezbedi traženi dokaz, kompajler će ubaciti poziv **materialize** makroa koji će izvršiti proveru da je statički tip nepromenljiv koristeći metodu *isImmutable*.

3.4 Jezički model

Jezički model se sastoji od struktura podataka, kao što su apstraktna sintaksna stabla, simboli ili tipovi koje koristi radni okvir za metaprogramiranje za predstavljanje: jezičke sintakse, specifikacije jezika itd. Kako se jezički model *scala.reflect*-a bazira na unutrašnjem mehanizmu Skalinog kompajlera, stoga je najbolji način za upoznavanje modela praćenje njegovog rada.

Prvi deo rada Skalinog kompajlera se odvija u dve faze: fazi parsiranja i fazi određivanja tipova. Nakon prve faze Skalin kompajler je izvršio samo parsiranje izvornih fajlova i još uvek nije ušao u proveru tipova. Osnovni gradivni element izlaza su čvorovi apstraktnog sintaksnog stabla. Pre svega, kompajler u potpunosti eliminiše sintaksna ulepšavanja (eng. *syntax desugaring*) i pritom nastaje normalizovani format

izvornog koda. Pored stabala taj kod sadrži i simbole koji su jedinstveni identifikatori koje *scalac* koristi da bi uspostavio veze između simbola i tipova.

Poslednja faza prevođenja koda je faza određivanja tipova, i nakon te faze svim stabilma su dodeljeni atributi i ona su spremna da uđu u proces koji će ih sažeti i pretvoriti u izvršni kod.

3.4.1 Stabla

U *scala.reflect*-u apstraktna sintaksna stabla (eng. AbstractSyntaxTree) su implementirana kao *case*-klase koje su nasleđene od apstraktne klase *Tree* koja obezbeđuje standardne funkcionalnosti. Čvorovi AST-a su, dakle klase čija deca su enkodovana u nepromenljivim poljima. Oni nemaju veze ka svojim roditeljima, što zahteva od programera da održava roditeljsku strukturu.

```
abstract class Tree extends Product {
  def pos: Position = {...}
  def setPos(pos: Position): this.type = {...}

  def tpe: Type = {...}
  def setType(t: Type): this.type = {...}

  def symbol: Symbol = {...}
  def setSymbol(s: Symbol): this.type = {...}

   ...

}
```

Neke jezičke funkcionalnosti nemaju pridružene AST čvorove. Neke od njih su enkodovane kao kombinacija jednostavnijih AST čvorova. Korišćenjem *scala.reflect*-a moguće je stvarati abstraktna sintaksna stabla koja narušavaju sintaksu jezika ili pravila koja se odnose na tipove. Konstruktori podataka su retko strogo tipizirani, pa je hijerarhija nasleđivanja otvorena. Ovaj dizajn je motivisan činjenicom da isto stablo treba da enkodira idiome na višim i nižim nivoima kroz različite stadijume kompilacije. Klasa *Tree* ima polje *position* koje može biti prazno ili označavati konkretne lokacije u odgovarajućim izvornim fajlovima (eng. *source files*). Podrazumevano, sva stabla imaju prazne pozicije, ali koje mogu biti određene pomoću *Tree.setPos* od strane standardnog parsera ili od strane programera. Pozicije mogu biti ofset pozicije ili intervalne. Mogu se koristiti da bi se aproksimativno odredilo poreklo stabla, ali su generalno nepouzdane, jer ne postoji 1-1 korespodencija između pozicija i stabala.

Stabla u *scala.reflect*-u mogu nostiti simbole i tipove koji predstavljaju semantičku informaciju.

3.4.2 Simboli

Svaki univerzum (klasa *Universe*) ima tabelu simbola koja predstavlja kolekciju pripadajućih definicija. Ova kolekcija može biti popunjena na nekoliko načina (unapred popunjena, učitana iz datoteke itd.), može biti strogo određena, a može i evoluirati tokom vremena. Zadatak popunjavanja table je prosleđen ogledalu. Tabela simbola se sastoji od simbola, tj. jedinstvenih identifikatora koji nose dodatne metapodatke. Jednakost simbola se bazira na jednakosti referenci, dok imena simbola nemaju udela u poređenju. Kako bi se proverilo da li data referenca pokazuje na određenu definiciju ili da li dve reference označavaju isto, potrebno je upoređivati njima dodeljene simbole. Ovaj postupak je ispravan samo u okviru istog univerzuma, dok je korelacija između objekata u različitim univerzumima nedefinisana.

```
abstract class Symbol(owner0: Symbol, pos0: Position, name0: Name) {
  def owner: Symbol = {...}
  def owner_=(o: Symbol): Unit = {...}

  def pos: Position = {...}
  def setPos(p: Position): this.type = {...}

  def name: Name = {...}
  def name_=(n: Name): Unit = {...}

  def info: Type = {...}
  def setInfo(info: Type): this.type = {...}
    ...

}
```

Simboli su organizovani u stablo po pripadnosti. Na primer, parametri pripadaju metodu, dok sam metod pripada klasi. Ovo stablo raste od *_root_* paketa. To znači da je tabela simbola suštinski stablo, ali je ime "tabela simbola" zaživelo u zajednici programera koji su razvijali kompajlere. Svaki simbol koji potencijalno poseduje druge simbole, npr. paket koji sadrži klase ili klasa koja sadrži metode, ima pridružen potpis (koji se čuva u *Symbol.info*) koji sadrži i promenljivu kolekciju svoje dece. Ove kolekcije se uglavnom lenjo izračunavaju kako kompajler ne bi morao da uključi sve njegove zavisnosti u tabelu simbola odjednom.

Simboli su jedini zvanični način pretraživanja strukture u kojoj je predstavljena definicija programa u *scala.reflect*-u. U većini slučajeva ovakav pristup daje tačan

prikaz strukture programa, ali neki scenariji se susreću sa znatnim problemima (npr. zbog činjenice da potpisi ne sadrže lokalne promenljive, nemoguće je proći kroz sve definicije korišćenjem isključivo tabele simbola).

3.4.3 Tipovi

Tipovi su najjednostavniji među osnovnim strukturama podataka *scala.reflect*-a. Za razliku od stabala ima manje različitih vrsta tipova, a za razliku od simbola tipovi nisu uređeni pomoću složene grafolike strukture.

```
abstract class Type {
  def termSymbol: Symbol = {...}
  def typeSymbol: Symbol = {...}
  def <:<(that: Type): Boolean = {...}
  def =:=(that: Type): Boolean = {...}
    ...
}
```

Takođe, za razliku od stabala i simbola, tipovi su nepromenljive strukture podataka. Stabla imaju promenljive atribute, simboli imaju promenljive potpise, a jedina promenljiva polja koja imaju tipovi su privatne keš memorije. Tipovi se stvaraju mnogo češće od stabala i simbola, stoga se oni agresivno keširaju i odgovarajuće keš memorije moraju biti promenljive. U jezičkom modelu tipovi imaju sledeće tri uloge: 1) određuju vrstu abstraktnih sintaksnih stabala, 2) preciziraju potpise simbola i 3) modeluju koncepte sistema tipova.

Određivanje tipova apstraktnih sintaksnih stabala

Sva stabla sa atributima imaju i dodeljene tipove. Kako bi se pristupilo datom tipu može se upotrebiti *Tree.tpe*.

Određivanje potpisa simbola

Većina simbola u *scala.reflect*-u imaju potpise koji se nalaze u $Symbol.info$. Ovi potpisi su takođe modelovani tipovima.

Modelovanje koncepta za sistem tipova

U svojim unutrašnjim mehanizmima, prevodilac koristi *Type* za proveru tipova, implicitno zaključivanje i izvedene tipove. Odatle se zaključuje da tip učauriva uobičajene pojmove Skalinog sistema tipova. Na primer, *Type*. <:<, predstavlja proveru

da li je nešto podtip, *Type.members* vraća listu članova, *Type.widen* vrši pretvaranje iz *singleton*-tipa u tip iz kojeg je izveden.

3.5 Zapisivanje jezičkog modela

Opisani model jezika čini osnovu *scala.reflect*-a ali rad sa njegovim strukturama podataka nije jednostavan. Često se dešava da jednostavni delovi koda budu prevedeni u duboko ugneždena apstraktna sintaksna stabla. Zbog toga, *scala.reflect* uključuje nekoliko olakšica koje su nalik domenski prilagođenim jezicima. Te olakšice se često realizuju upotrebom navoda (eng. quote) u okviru kojih jezička sintaksa smeštena u određeni kontekst se koristi kao DSL koji proizvodi odgovarajući element jezičkog modela.

3.5.1 Reify

Kao dodatak klasi *Tree*, koja ne poseduje statički tip, *scala.reflect* podržava *Expr*[*T*] koji predstavlja statički tipiziarni omotač klase *Tree*.

```
1 trait Expr[+T] {
2   def tree: Tree
3   def splice: T
4   val value: T
5   ...
6 }
```

Kao što se može videti u uprošćenoj definiciji *Expr*, odgovarajućem stablu se može pristupiti korišćenjem metode *Expr.tree*.

Instance tipa *Expr* mogu biti kreirane ili pomoću argumenata koji čine apstraktno sintaksno drvo i pridruženi tipovi ili automatski pozivanjem metode *reify* koju obezbeđuje *scala.reflect*. Prvi pristup je zamoran i može narušiti statičnost tipova pa taj način nije preporučen (kako je ovaj pristup prvi nastao, zadržao se radi kompatibilnosti sa pređašnjim verzijama biblioteke).

reify je metod koji kao argument prima tipiziran izraz tipa *T* a kao rezultat vraća ekvivalent tipa *Expr*[*T*]. Glavna svrha ovog metoda je reprodukcija sintaksne strukture datog izraza koja će se dalje ispitivati za vreme izvršavanja. Ovo se dešava u okviru granica koje podržava jezički model - što znači da *reify* izvodi eliminisanje sintaksnih ulepšavanja, ne čuva komentare, ni formatiranje izvornog koda.

Kada je reč o zapisivanju sintaksnih stabala, *reify* se pokazao efikasnim u slučaju manjih delova koda, ali pošto su u metaprogramiranju zahtevi korisnika često veći, uočeni su sledeći nedostaci:

1. Zahteva se dodatna disciplina po pitanju tipova. Instance tipa *Expr* su zahtevnije za prosleđivanje nego obična sintaksna stabla zato što su statički tipizirane. Kao rezultat, metaprogrami koji koriste *reify* su opširniji nego njihovi ekvivalenti koji koriste stabla.

2. Metaprogrameri mogu samo da koriste tipizirane delove koda, a zbog rasta u dužini metaprograma postaje neophodno podeliti kod na više delova.

3. *reify* sprečava modularizaciju zato što je nemoguće razdvojiti definicije i reference na njih u različite navode.

4. Ima ograničene mogućnosti slaganja (kompozicije).

5. Podržava samo konstrukciju, pozivi metoda se ne mogu koristiti na poziciji šablona (eng. *pattern position*) što znači da *reify* može biti korišćen samo za konstrukciju ali ne i dekonstrukciju sintaksnih stabala.

3.5.2 Kvazinavodi

Kako se inicijalni plan zapisivanja apstraktnih sintaksnih stabala pomoću *reify* metode pokazao nedovoljnim za praktičnu upotrebu *scala.reflect*-a, bilo je potrebno novo rešenje. Do njega se došlo u vidu kvazinavoda, koji su mnogo fleksibilniji i mogu se koristiti za prepoznavanje šablona. *Denys Shabalin*[9] je implementirao prototip kvazinavoda koji je uvršćen u Scala 2.11.

Sintaksni aspekt: Kvazinavodi predstavljaju familiju interpolatora koja pokriva: iskaze, tipove, slučajeve, šablone i enumeratore (u primeru ispod su prikazani njihovi zapisi u Skali). Neki sintaksni elementi (npr. import klauzula) nemaju pridružena stabla, pa ne mogu biti izraženi pomoću kvazinavoda.

```
1  import scala.reflect.runtime.universe._
2
3  import scala.reflect.runtime.universe._
4
5  val res0 =  q"List(42)"  // res0: Tree = List(42)
6
7  val res1 = tq"List[Int]"  // res1: Tree = List[Int]
8
9  val res2 =  cq"List(x) => x" // res2: CaseDef = case List((x @ _)) => x
10
```

```
11 val res3 =  pq"List(x)" // res3: Tree = List((x @ _))
12
13 val res4 =  fq"x <- List(42)" // res4: Tree =  <-((x @ _), List(42))
```

Kompozicija stabala se vrši pomoću ugrađene funkcionalnosti interpolacije niski.

Semantički aspekt: Kvazinavodi dopuštaju loše tipizirane delove koda. Kao rezultat, nejasno je kako se pridružuje semantička informacija stablima kreiranim pomoću kvazinavoda, jer kod stavljen pod navode nema semantičku informaciju. To je navelo programere koji su radili na *scala.reflect*-u na špekulativnu proveru, koja bi se u slučaju uspeha pridruživala kao semantička informacija rezultatu. Ovakav pristup je naišao na poteškoće. Zbog eliminacije sintaksnih ulepšavanja *scala.reflect* teško uspostavlja veze između tipiziranih i netipiziranih stabala. Male izmene tipova u delovima koda pod navodima, mogu imati veliki uticaj na rezultat. Navedeni problemi su za krajnji rezultat imali odustajanje od pridruživanja semantičke inofrmacije.

3.5.3 Oznake za tipove

TypeTag je tipska klasa čije instance prave omot oko klase *scala.reflect.Type* koja predstavlja njihov parametar za tip. Glavna upotreba oznaka za tip je prenos reprezentacije parametara za tip metaprogramima koji se izvršavaju u vreme kompajliranja. Slično klasi *Expr*, oznake za tip može samostalno stvarati programer, ili se osloniti na automatsko izvodjenje koje obezbeđuje *scala.reflect*. Prvi pristup je podložniji greškama, pa se drugi primenjuje u praksi.

Kada se zahteva instanca klase TypeTag, a nema nijedne u oblasti definisanosti, Skalin kompajler će je generisati. Navedeni proces u ovoj sekciji će se zvati materijalizacija.

Scala.reflect obezbeđuje dve vrste oznaka za tip: 1) WeakTypeTag i 2) TypeTag. Prvi može reifikovati (opredmetiti, u ovom slučaju to znači da može biti omot svakog tipa) sve tipove, dok drugi to radi samo u slučaju tipova koji nemaju reference na neoznačene tipske parametre ni apstraktne tipske članove, inače će doći do statičke greške. Ova razlika je od pomoći u slučaju lanaca poziva generičkih metoda, jer garantuje korektnu propagaciju tipova.

Oznake za tip čuvaju semantičku informaciju koristeći sličnu strategiju kao *reify* , sa slobodnim promenljivama koje se pojavljuju u vidu simbola. Stoga, implementacija oznaka treba da serializuje aproksimaciju relevantne sekcije tabele simbola, što dovodi do parcijalno tačnih rešenja. Oznake za tip su se pokazali korisnim, jer u nekim

slučajevima dovode do pojednostavljenja koda koji manipuliše sa tipovima. Nažalost, oblast primenljivosti im je ograničena, jer ne rade dobro sa složenim tipovima i ne podržavaju prepoznavanje šablona. Tipovi u *scala.reflect*-u moraju sadržati semantičku informaciju, pa se dalji razvoj oznaka za tipove pokazao nužnim.

4 Makroi u programskom jeziku Skala

4.1 Def makroi

Def makroi su metode čiji pozivi se konkretizuju u vreme kompilacije, tj. prevode u kod koji barata apstraktnim sintaksnim stablima. Tokom proširivanja, metaprogram asociran sa makroom transformiše AS drvo koje predstavlja primenu makroa u drugo AS drvo. Takvi metaprogrami operišu sa kontekstom, koji prikazuje kod koji treba da se proširuje i sa ogledalom koje vraća program koji će se kompajlirati.

Ova funkcionalnost je dostupna u Skali kao eksperimentalno svojstvo od verzije v2.10 i izgrađena je nad funkcionalnostima koje obezbeđuje *scala.reflect* objašnjen u prethodnom poglavlju [8].

4.1.1 Motivacija

Def makroi su inspirisani makroima iz programskog jezika Nemerle (koji su sa druge strane inspirisani makroima iz Haskela i Scheme-a). Kao rezultat, to je dovelo do sledećih ciljeva u dizajnu:

- Minimizovati promene sintakse jezika (u vreme nastanka def makroa, Skala je već postojala deset godina). Stoga se išlo na dodavanje što manjeg broja svojstava kako bi se složenost jezika držala pod kontrolom. Makroi koji izgledaju kao regularne metode su se slagali sa ovakvim pristupom.

- Izbegavati fragmentaciju jezika. Kako bi se sprečilo da def makroi razdvoje jezik na makro-zasnovane dijalekte, argumenti makroa i proširivanje makroa moraju biti dobro tipizirani. To je piscima makroa onemogućilo menjanje sintakse jezika, ali je s druge strane korisnicima makroa obezbedilo da razumeju ponašanje makroa na osnovu potpisa tipova.

- Dopustiti izvršavanje proizvoljne logike tokom kompilacije. Iako postoje sistemi koji ograničavaju metaprograme na jezike koji nisu Tjuring kompletni, ovde je cilj bio da dizajn bude manje ograničen kako bi se povećala izražajnost i prihvatiljivost.

- Obezbediti pristup i sintaksnim i semantičkim informacijama o programu. Današnji sistemi uglavnom dopuštaju sintaksne transformacije programa. Korišćenje Skalinog sistema tipova je omogućio makroima pristup semantici.

4.1.2 Prikaz sistema za rad sa makroima

U ovoj sekciji, razmatra se makro koji implementira podskup funkcionalnosti LINQ-a (eng. *Language Integrated Query*) radi prikaza komponenti makro sistema na višem nivou.

LINQ je tehnika kojom se postiže glatka integracija rada sa bazom podataka i programskog jezika. Čest pristup ovoj tehnici se sastoji u predstavljanju izvora podataka kao kolekcija (koje su prisutne među tipovima u posmatranom jeziku), time dopuštajući korisniku da piše upite kao nizove poziva ka ovim tipovima koristeći poznate metode višeg reda kao što su *map*, *filter* i druge.

Ispod je definisana crta $Query[T]$ koja učauriva upit koji vraća kolekciju objekata tipa T. Pored nje, definisane su i njeni naslednici *Table* i *Select* koji predstavljaju određene tipove upita. *Table* je namenjen za izvor podataka koji poznaje osnovni tip, a *Select* modeluje ograničeni podskup SQL SELECT rečenica pomoću jednostavnog $NodeAST$-a.

```
1 trait Query[T]
2   case class Table[T: TypeTag]() extends Query[T]
3   case class Select[T, U](q: Query[T], fn: Node[U]) extends Query[U]
4 trait Node[T]
5   case class Ref[T](name: String) extends Node[T]
6   object Database {
7   def execute[T](q: Query[T]): List[T] = { ... }
8 }
```

U ovom modelu, SQL upit 'SELECT *name* FROM *users*' je predstavljen sa $Select(Table[User](), Ref[String]("name"))$. Upiti poput ovog se mogu izvršavati sa *Database.execute* koji ih prevodi u SQL, koji potom šalje bazi podataka, prihvata odgovor koji potom prevodi u objektne podatke. Glavni aspekt LINQ olakšica je prikladan zapis upita. Kako je SQL, kao i svaka reprezentacija zasnovana na niskama, sklon sintaksnim greškama, greškama u tipovima i ubacivanjima, nijedan od načina nije dovoljno svrsishodan. Eksplicitno instanciranje *Query* objekata veoma je opširno, pa ipak ne obuhvata sve tipove grešaka.

U sledećem nacrtu, definiše se LINQ API u formi metoda koje prihvataju kolekcioni API iz standardne biblioteke. Ideja je bila da korisnici budu u mogućnosti

da enkoduju upite u intuitivne i statički tipizirane pozive ka API-ju, koje će potom biblioteka prevesti u pozive Query konstruktora.

```
object Query {
  implicit class QueryApi[T](q: Query[T]) {
    def map[U](fn: T => U): Query[U] = { ... }
  }
}
case class User(name: String)
    val users = Table[User]()
    users.map(u => u.name)
// prevedeno u : Select(users, Ref[String]("name"))
```

Željeni efekt se može postići pomoću metaprogramiranja u vreme kompajliranja. Metaprogram koji se izvršava u vreme kompajliranja može da detektuje sve pozive *Query.map* i da ih prezapiše u pozive *Select* metoda sa parametrima za *map* transformisanim u odgovarajuće instance *Node*-a.

```
import scala.language.experimental.macros
  object Query {
    implicit class QueryApi[T](q: Query[T]) {
    def map[U](fn: T => U): Query[U] = macro QueryMacros.map
  }
}
```

QueyApi.map se naziva makro def. Kako su makroi eksperimentalno svojstvo jezika, za njihovo definisanje je neophodno imati import **import scala.language. experimental.macros** u leksičkoj oblasti važenja definicije ili obezbediti odgovarajuće podešavanje u kompajlerskim flegovima. Def makroi izgledaju kao normalne metode, u smislu da mogu imati termove kao parametre, tipske parametre i povratne tipove. Poput regularnih metoda, def makroi mogu biti deklarisani unutar ili izvan klasa, mogu biti monomorfni ili polimorfni, i mogu učestvovati u inferenciji tipova i implicitnim pretragama. Tela def makroa imaju neuobičajenu sintaksu, počinju sa ključnom rečju *macro* koju prati identifikator koji referiše na implementaciju makroa, ascoirani metaprogram koji izvršava kompajler kada naiđe na odgovarajuću primenu makroa. Implementacije makroa uzimaju kontekst kompajlera koji predstavlja ulaznu tačku u makro API. Makro API se sastoji od metaprogramerskog skupa opšte namene obezbeđenog od *scala.reflect*-a i nekoliko specijalizovanih olakšica koje se odnose na ekspanziju makroa. Tipična prva linija implementacije makroa je *import c.universe._*koja čini da ceo *scala.reflect* API bude dostupan metaprogrameru. Kao

dodatak kontekstu, za svaki termovski parametar def makroa, njegova makro implementacija prihvata termovski parametar koji nosi reprezentaciju odgovarajućeg argumenta makro aplikacije. Implementacija makroa vraća apstraktno sintaksno drvo, i ovo AST zamenjuje originalnu primenu makroa u kompilacijskoj protočnoj obradi (eng. pipeline).

Mogućnost def makroa da pristupaju tipovima značajno unapređuje korisničko iskustvo u poređenju sa čistim sintaksnim prevođenjem. Prvo, čak i pre nego što se *Query.map* proširi, kompajler proverava njegov argument, time garantujući da su upiti dobro tipizirani. Drugo, postoji način da se pouzdano provere oblici podržanih lambda.

4.1.3 Def makroi i implementacija makroa

Def makroi su razdvojeni u dva dela: 1) definicije makroa koji obezbeđuju tipske signature za argumente i proširenja 2) implementacije makroa koje obezbeđuju metaprograme koji se izvršavaju nad reprezentacijom termova i tipovima argumenata makro aplikacije. Ova dva dela su uvezana zajedno preko **macro impl** reference, jezičkog kontrukta koji čini telo def makroa. Rezultat implementacije makroa je novo AST, koje će biti uvučeno na mestu poziva uz proveru tipa.

U sledećem primeru je naveden deo koda sa def makroom i njegovom implementacijom.

```
1  class FastArrayParsers[T] extends BaseParsers[T, Array[T]] with
       RepParsers with FlatMapParsers {
2    def apply(rules: => Unit): FinalFastParserImpl = macro ArrayParserImpl
       .ArrayParserImpl[T]
3  }
4
5  /**
6   * Implementacija Parser kombinatora koja se bavi Array[T] ulazima
7   */
8  object ArrayParserImpl {
9    def ArrayParserImpl[T: context.WeakTypeTag](context: Context)(rules:
       context.Tree): context.Tree = {
10     new BaseImpl with RulesTransformer with RulesInliner
11       with ParseRules with BaseParsersImpl with RepParsersImpl
12       with FlatMapImpl with RuleCombiner
13       with ArrayInput with DefaultParseError with IgnoreResults {
14
15       val c: context.type = context
16
17       import c.universe._
18
19       def inputElemType = c.typecheck(tq"${implicitly[c.WeakTypeTag[T]]}
       ",c.TYPEmode).tpe
```

```
20
21      }.FastParser(rules)
22    }
23  }
```

Definicije makroa izgledaju kao regularne metode, s tim što imaju specijalna tela. Imaju regularne potpise, mogu biti definisane na regularnim lokacijama u kodu, i mogu imati većinu regularnih modifikatora.

Implementacije makroa su metode koje mogu sadržati proizvoljan Skala kod i mogu se pozivati iz proizvoljnog koda. Postoji ograničenje da implementacije makroa moraju biti public, static i da ne smeju biti preopterećene (eng. overloaded), koje je uvedeno sa namerom da se sistemu za rad sa makroima olakša ulaz u implementaciju.

Definicije makroa i implementacije makroa moraju biti kompatibilne jedna sa drugom, što suštinski znači da za svaki termovski parametar def makroa, njegova implementacija mora imati odgovarajući parametar tipa *Tree* ili *Expr* iz *scala.reflect* API-ja. Takođe, za svaki tip argumenta iz reference implementacije makroa, data implementacija mora imati odgovarajući tipski parametar kao i opcioni implicitni parametar tipa *TypeTag*. Finalno, povratni tipovi moraju da odgovaraju jedni drugima slično kao termovski parametri.

Nabrajanje ispod daje neke primere funkcionisanja kompatibilnosti. U datom kodu, svi parovi istoimenih definicija makroa i implementacija su kompatibilni.

```
1  trait BaseParsers[Elem, Input] {
2    ...
3    def range(a: Elem, b: Elem): Parser[Elem] = ???
4
5    def accept(p1: ElemOrRange, p2: ElemOrRange*):Parser[Elem] = ???
6
7    def acceptIf(f: Elem => Boolean): Parser[Elem] = ???
8
9    def wildcard: Parser[Elem] = ???
10   ...
11
12 }
13
14 trait BaseParsersImpl{
15  import c.universe._
16   ...
17   private def parseRange(a: c.Tree, b: c.Tree, rs: ResultsStruct):
      BaseParsersImpl.this.c.universe.Tree =
18   {
19     q"""
20      if ($isNEOI && $currentInput >= $a && $currentInput <= $b){
21       ${rs.assignNew(currentInput, inputElemType)}
22       $advance
```

```
23          $success = true
24        }
25        else {
26           $success = false
27           ${pushError("expected in range ('" + show(a) + "', '" + show(b)
      + "')", pos)}
28        }
29       """
30    }
31
32    private def parseAccept(a: List[c.Tree],negate: Boolean,rs:
       ResultsStruct): BaseParsersImpl.this.c.universe.Tree =
33    {
34       val ranges = if (negate)  q"!(${getAcceptedElem(a)})"
35                    else q"(${getAcceptedElem(a)})"
36       q"""
37          if ($isNEOI && $ranges){
38            ${rs.assignNew(currentInput, inputElemType)}
39            $advance
40            $success = true
41           }
42           else {
43              $success = false
44              ${pushError("expected element in " + a.map(prettyPrint(_)).
      mkString, pos)}
45           }
46       """
47    }
48
49
50    private def parseAcceptIf(f: c.Tree, rs: ResultsStruct):
       BaseParsersImpl.this.c.universe.Tree=
51    {
52       q"""
53         if ($isNEOI && $f($currentInput)){
54          ${rs.assignNew(currentInput, inputElemType)}
55          $advance
56          $success = true
57         }
58         else {
59           $success = false
60           ${pushError("acceptIf combinator failed", pos)}
61         }
62         """
63    }
64
65    private def parseWildcard(rs: ResultsStruct): BaseParsersImpl.this.c.
       universe.Tree=
66    {
67       q"""
68          if ($isNEOI){
69            ${rs.assignNew(currentInput, inputElemType)}
70            $advance
71            $success = true
```

```
72          }
73          else
74            $success = false
75
76        """
77    }
78    ...
79  }
```

4.1.4 Termovski parametri

Prvi parametar implementacije makroa mora biti kontekst koji obezbeđuje pristup makro API-ju. Kontekst je omotač oko *scala.reflect* univerzuma u vreme kompilacije. Kao dodatak standardnom *scala.reflect API*-ju, konteksti takođe sadrže neke dodatne funkcionalnosti koje imaju smisla samo u makroima (primeri su $Context.prefix$ ili $Context.abort$) pa stoga nisu uključene u deo opšte namene *scala.reflect*-a. Pored toga, za svaki parmetar def makroa, implementacija makroa mora imati parametar koji će čuvati AST odgovarajućeg argumenta makro aplikacije. Ovi parametri mogu biti ili tipa $Expr$ ili tipa $Tree$. Tokom proširivanja makroa, sistem za rad sa makroima napraviti odgovarajuće argumente za implementaciju makroa.

Sposobnost implementacija makroa da rade sa izrazima je zastarela funkcionalnost nasleđena od inicijalnog izdanja def makroa, kada je jedina podrška za AST bila $reify$ koja je radila samo sa statički tipiziranim stablima, tj izrazima. Stoga, implementacije makroa su prihvatale izraze kao parametre i vraćale izraze radi konzistentnosti, jer je jedino tako bilo moguće koristiti $reify$. Međutim, danas većina metaprogramera koristi kvazinavode koji rade sa stablima. Kada je omogućeno da stablo bude argument kao i povratna vrednost implementacije makroa, potpisi metoda su se pojednostavili, ali su zadržani izrazi zbog kompatibilnosti sa pređašnjim verzijama biblioteke.

Prefiks vezan za primenu makroa, ekvivalent *this* za def makro, nije predstavljen kao eksplicitni parametar def makroa, pa takođe nije predstavljen parametrom u implementaciji makroa. Umesto toga, postoji makro API $Context.prefix$ koji obezbeđuje odgovarajući izraz. Ekvivalentno stablo se može dobiti pozivom $Expr.tree$ nad datim izrazom.

Kako bi se izbegla zabuna, zahteva se stroga korespodencija između parametara definicija makroa i implementacija. Za svaki parametar def makroa, mora postojati

istoimeni parametar u implementaciji (osim u slučaju kompajler-generisanih parametara sintetisanih za neka svojstva jezika, kada je implementaciji makroa dopušteno da izabere proizvoljno ime).

Koncept korespodencije parametara je bitan deo algoritma za proveru tipova u sistemu za rad sa makroima. Kako implementacije makroa mogu za argumente imati izraze, treba osigurati da tipovi parametara odgovaraju statičkim tipovima izraza. Na primer, ako implementacija makroa sadrži parametar $x : c.Expr[Int]$, onda i definicja makroa mora imati odgovarajući parametar $x : Int$. Međutim, sprovođenje u delo ovog koncepta je zahtevno. Intuitivno, izgleda da je dovoljno proveriti **Expr** koji je parametar implementacije makroa i uporediti rezultat sa odgovarajućim tipom def makroa, ali to nije tako jednostavno. Oblasti važenja def makroa i odgovarajućih implementacija se razlikuju, pa jednostavna provera tipova ne radi. Iz tog razloga, algoritam za proveru tipova pored provere jednakosti struktura datih tipova, ima tri specijalna pravila radi prevazilaženja potencijalnih razlika nastalih usled različitih oblasti važenja:

1. Tipski argumenti reference na makro implementaciju odgovaraju tipskim parametrima makro implementacije. To omogućava implementaciji makroa da referiše na tipove koji su lokalni za oblast važenja odgovarajuće definicije makroa.

2. Tip klase u kojoj je sadržana definicija makroa odgovara $c.PrefixType$ u implementaciji makroa, gde je c parametar u kom se nalazi kontekst. Iako se upotreba ovog pravila može izraziti pomoću prethodnog, ono omogućava mnogo lakše referisanje na tipove unutar klase koja koristi *cake pattern*[10].

3. Za def makro parametar pd i parametar implementacije pi, $pd.U$ odgovara $pi.value.U$. Kad god je pd jednako T, pi je predviđeno da bude $Expr[T]$, tip $pd.U$ zavisan od putanje referiše na istu definiciju kao i $pi.value.U$. Time je omogućeno da makro implementacije odgovaraju definicijama makroa koje imaju zavisne tipove metoda.

U sledećem primeru, nalazi se skica reimplementacije $reify$ metode is $scala.reflect$ radnog okvira. Kod prikazuje uprošćenu verziju $MyUniverse$ *cake*-a, kao i $MyExpr$ parče koje predstavlja $MyExpr[T]$ (linija 10). Unutar $MyUniverse$ definiše se reify kao def makro (linija 13), a potom je ispisan potpis zasnovan na izrazima za njegovu makro implementaciju (linija 17).

```
1  import scala.language.experimental.macros
2  import scala.reflect.macros.blackbox.Context
3
4   trait MyExprs {
5    self: MyUniverse =>
6
7    trait MyExpr[T] { ... }
8   }
9
10  trait MyUniverse extends MyExprs
11   with ... {
12
13   def reify[T](expr: T): MyExpr[T] = macro MyUniverseMacros.reify[T]
14  }
15
16  object MyUniverseMacros {
17   def reify[T]
18   (c: Context{ type PrefixType = MyUniverse })
19     (expr: c.Expr[T]):
20       c.Expr[c.prefix.value.MyExpr[T]] = { ... }
21  }
```

Glomazna signatura na linijama 17-20 se odnosi na korespodenciju između def makroa i implementacije makroa. Glavni izazov je izražavanje zavisnosti putanje rezultujućeg tipa na osnovu tipa prefiksa makro aplikacije. Da bi se to postiglo, na liniji 18 se nalazi tip prefiksa (pravilo 2), a potom se koristi tip prefiksa u $c.prefix.value$ na liniji 19 (pravilo 3).

4.1.5 Tipski parametri

Pored pristupa termovskim argumentima makro aplikacije, implementacije makroa mogu pristupiti reprezentaciji tipova asociranih sa makro aplikacijom. Metaprogrameri su najviše zainteresovani za tipove argumenata makro aplikacije, ali takođe postoje načini da se pristupi tipovima argumenta ograđujućih klasa i crta.

Za svaki tip argumenta reference na makro implementaciju, makro implementacija mora imati tipski parametar. Tokom ekspanzije makroa, takav tipski parametar predstavlja odgovarajući tip argumenta reference na makro implementaciju, u kojoj su upotrebe tipskih parametara iz def makroa, tipovi parametara ograđujućih klasa itd. zamenjeni sa njihovim instancijacijama u odgovarajućim makro aplikacijama.

Kad god je metaprogramer zainteresovan za proveru ugnježdenog tipa određenog tipskog parametra, trebalo bi da deklariše implicitnu oznaku za tip za taj ugnježdeni

tip. Tokom ekspanzije makroa, kompajler će automatski izračunati relevantne tipove i proslediti ih u makro implementaciju umotane u oznaku za tip.

```
class FastArrayParsers[T] extends BaseParsers[T, Array[T]] with
    RepParsers with FlatMapParsers {
  def apply(rules: => Unit): FinalFastParserImpl = macro ArrayParserImpl
    .ArrayParserImpl[T]
}

/**
 * Implementacija Parser kombinatora koja se bavi ulazima tipa Array[T]
 */
object ArrayParserImpl {
  def ArrayParserImpl[T: context.WeakTypeTag](context: Context)(rules:
    context.Tree): context.Tree =  {
     new BaseImpl with RulesTransformer with RulesInliner
       with ParseRules with BaseParsersImpl with RepParsersImpl
       with FlatMapImpl with RuleCombiner
       with ArrayInput with DefaultParseError with IgnoreResults {

       val c: context.type = context

       import c.universe._

       def inputElemType = c.typecheck(tq"${implicitly[c.WeakTypeTag[T]]}
    ",c.TYPEmode).tpe

     }.FastParser(rules)
  }
}
```

Na primeru parsera:

```
val celobrojniNiz = FastParser{
    def niz = lit("[")  ~> repsep(neki, ",") <~ "]"

    def broj = number ^^(_.toString.toInt)
  }

  println(celobrojniNiz.niz("[1,2,3,4,5]"))
```

tokom ekspanzije makroa zaduženog za celobrojni niz, *celobrojniNiz*.

niz("[1, 2, 3, 4, 5]"), u telu implementacije makroa *inputElemType* će biti tip koji predstavlja *Int*.

Kako bi se zapis učinio konciznijim, u praksi metaprogrameri uglavnom deklarišu implicitne paramere preko kontekstnih granica, primer $ArrayParserImpl[T : context.WeakTypeTag]](...) = ...$ U tom slučaju implicitnim parametrima se može pristupiti pomoću $implicitly[c.WeakTypeTag[...]]$ ili preko $weakTypeOf[...]$, pomoćne

funkcije definisane u $Scala.reflect$ koja prihvata implicitnu oznaku za tip i odmotava ga.

Oznake za tip su indirektni način da se pristupi tipu, ali su slično kao i izrazi tu iz razloga kompatibilnosti sa pređašnjim verzijama. Metoda $reify$ je statički tipizirana, pa ne može raditi sa običnim tipovima i zahteva statički tipizirane oznake za tip. Inicijalno, $reify$ je trebalo da bude glavna metoda za rad sa makroima, rešeno je da se koriste oznake za tip, ne tipovi u potpisima implementacije makroa. Za razliku od izraza, nije se došlo do jednostavnijeg pristupa tipovima u implementacijama makroa, a to je jedan od ciljeva u daljem razvoju makroa u Skali.

4.1.6 Razdvojena kompilacija

Primena makroa i njegova implementacija se moraju nalaziti u različitim kompilacijskim jedinicama, zbog činjenice da trenutno ne postoji dovoljno pouzdana tehnologija za interpretaciju ili JIT-kompilaciju AST-ova. U vreme pisanja, jedini način da kompajler pozove implementaciju makroa je preko JVM refleksije prekompajliranog bajtkoda. Ako se makro apilikacija kompajlira zajedno sa odgovarajućom implementacijom makroa, u tom trenutku jos uvek nema bajtkoda za tu implementaciju makroa, što za tekući sistem za rad sa makroima čini ekspanziju makroa nemogućom.

Treba primetiti da ovo ograničenje ne znači da se definicije makroa i njihove primene moraju kompajlirati odvojeno. Ako implementacija makroa potiče iz različite kompilacijske jedinice, tada korisnici mogu koristiti definicije makroa u istim kompilacijskim jedinicama u kojima su definisane. Kao posledica, ovakav makro sistem dopušta makroe koji generišu druge makroe.

Pored toga, razdvojena kompilacija je ograničenje koje je vidno uticalo na Skalino radno okruženje. Česta situacija gde se ograničenje pokazalo ozbiljno neugodnim, je kada autori projekta žele da definišu domenski specifične pomoćne funkcije. Kako bi to postigli, primorani su da projekat razdvoje na dva dela: 1) makro definicije i njihove tranzitivne zavisnosti iz originalnog projekta 2) ostatak originalnog projekta koji zavisi od prvog dela i koristi makroe definisane u njemu.

Srećom, ovo ograničenje ne ometa druge radne tokove zasnovane na makroima. Nema problema za regularne korisnike kada zavise od biblioteka treće strane koje koriste makroe, jer se biblioteke treće strane po definiciji prekompiliraju. Takođe, nema potrebe da se kreiraju razdvojeni projekti samo da bi se testirali makroi, jer svi sistemi za izgradnju u Skalinom okruženju kompajliraju odvojeno projekte i njihove testove. Još se može dodati i da je svaka komanda u Skalinom REPL-u procesirana u različitoj kompilacijskoj jedinici, što dopušta razvoj makroa na interaktivan način.

Plan daljeg razvoja je definitivno okrenut otklanjanju ograničenja razdvojene kompilacije, jedan od pokušaja je razvoj biblioteke *scala.meta*[11].

4.1.7 Ekspanzija makroa

Ekspanzija makroa je inicijalno zamišljena kao jedan od koraka u procesu provere tipova.

Def makroi su podeljeni u dve grupe, *Whitebox* i *Blackbox* makroe, a podela je izvršena na osnovu toga kako interaguju sa proverom tipova. *Whitebox* makroi ne izvršavaju samo prezapisivanje stabala, već mogu uticati na implicitnu pretragu i inferenciju tipova. *Blackbox* makroi samo prezapisuju stabla, pa proveravač tipova na njih gleda kao na crne kutije (tako su i dobili imena). U Skalinoj verziji 2.10 svi makroi su whitebox, dok su u verziji v2.11 ubačeni blackbox makroi.

Ekspanzija makroa oba tipa funkcioniše na sličan način. Osim ako drugačije nije navedeno, opis protočne obrade ekspanzije se primenjuje na oba tipa. Eksplicitno će biti naglašene situacije kada tip makroa menja kako se posao odvija u proveravaču tipova.

4.1.7.1 Protočna obrada ekspanzije

Proveravač tipova izvodi ekspanziju tipova kada naiđe na primenu definicije makroa. Do toga može doći kada se aplikacije pišu od strane programera ili kada ih sintetiše kompajler (primer je umetanje implicitnog argumenta ili implicitnih konverzija).

Kada proveravač tipova naiđe na datu primenu metode, započinje proces zasnovan na implementaciji koji se sastoji u proveri tipa reference na metodu, opcionom odlučivanju u slučaju da postoji preopterećenje metoda, proveri tipova argumenata, opcionoj inferenciji implicitnih argumenata i opcionoj inferenciji tipskih argumenata. Makro ekspanzije se obično dešavaju nakon završetka svih ovih koraka, kako bi se osiguralo da implementacija makroa može da radi sa konzistentnim oblikom makro primene.

Blackbox makroi uvek rade ovako. *Whitebox* makroi se takođe mogu proširiti kada inferencija tipova ne može da donese zaključak o nekom od tipskih argumenata u makro aplikaciji. Kao rezultat odluke da se provere tipovi reference na metod i argumenata metode pre makro ekpanzije, proširivanja rade iznutra ka spolja. To znači da se zagrađujuća makro aplikacija uvek proširuje nakon zagrađujućih makro aplikacija koje se nalaze u prefiksima ili argumentima. Posledica toga je da makro ekspanzije uvek vide svoje prefikse i argumente potpuno proširene.

Tokom ekspanzije makroa, makro aplikacija je destruktuirana i sistem za rad sa makroima skuplja termovske i tipske argumente za odgovarajuću implementaciju makroa, prema pravilima opisanim u sekcijma 4.1.4 i 4.1.5. Zajedno sa kontekstom, ovi argumenti su prosleđeni implementaciji makroa.

Definicije makroa prihvata puno ime implementacije makroa, koje je potom iskorišćeno za dinamičko učitavanje zagrađujuće klase pomoću JVM refleksije (Skalin kompajler se izvršava na JVM, pa se JVM koristi za pokretanje implementacija makroa). Nakon učitavanja klase, kompajler poziva metod koji sadrži kompajlirani bajtkod implementacije makroa. Uprkos tome što je nesputano izvršavanje koda bio jedan od dizajnerskih ciljeva, ono može imati štetne efekte kao što su usporenja ili zaustavljanja kompajlera, pa je ovo nešto na čemu treba napredovati u budućnosti.

Sve makro ekspanzije u istoj kompilacijskoj jedinici dele isti JVM učitavač klase, što znači da dele globalno promenljivo stanje. Konrtolisanje bočnih efekata uz simultano obezbeđivanje prostora za komunkaciju između makroa je takođe jedan od bitnijih zadataka u nekoj od narednih verzija Skale.

Pozivanje makro implementacije se završava na jedan od tri moguća načina: 1) regularan povratak, gde je povratna vrednost ekspanzija makroa, 2) namerni prekid pomoću *c.abort*, 3) neprerađen izuzetak. U prvom slučaju eskpanzija se nastavlja, inače kompajler vraća grešku i ekspanzija se završava.

Nakon uspešnog završetka pozivanja makro implementacije, rezultujuća ekspanzija prolazi kroz proveru povratnog tipa definicije makroa da bi se osigurala bezbedna integracija u program koji se kompajlira. Ova provera tipa može dovesti do rekurzivnih ekspanzija makroa, što može rezultovati prepunjavanjem steka, ako rekurzija ide mnogo u dubinu ili se ne završava. To je jedan od većih nedostataka trenutnog sistema za rad sa makroima.

Na kraju, ekspanzija makroa koja je uspešno prošla proveru tipova zamenjuje makro aplikaciju u kompajlerovom AST. Dalja provera tipova i faze u kompajliranju će raditi nad proširenim makroom, umesto sa originalnom makro aplikacijom.

Zaključak je da je makro ekspanizja u ovom dizajnu komplikovan proces usko vezan sa proverom tipova. Sa jedne strane, značajno je zakomplikovana priča o makro ekspanziji, a sa druge korišćene su jedinstvene tehnike što je jedno od načela otvorenog koda Skaline zajednice.

4.1.7.2 Dejstvo na implicitnu pretragu

Implicitna pretraga je podsistem proveravača tipova koja je aktivirana kada primeni metoda nedostaje implicitni argument ili implicitna konverzija zahteva term u

njegovom očekivanom tipu.

Kada započinje implicitna pretraga, proveravač tipova pravi listu implicitnih kandidata (npr. implicitni *val*-ovi i *def*-ovi koji su dostupni u oblasti važenja). Sledeće, za ove kandidate se spekulativno proveravaju tipovi u redosledu definisanom implementacijom, da bi se utvrdilo da li odgovaraju parametrima pretrage. Na kraju, preostali kandidati se upoređuju jedni sa drugima prema imlicitnom algoritmu rangiranja, i najbolji se biraju kao rezultat pretrage. Ako nema primenljivih kandidata, ili ih ima više sa istim rangom, implicitna pretraga vraća grešku.

Kako implicitna pretraga uključuje proveru tipova, na nju može uticati ekspanzija makroa. U Skali 2.10, dopuštena je ekspanzija makroa tokom provere ispravnosti implicitnih kandidata. Odatle, implicitni makroi su imali mogućnost dinamičkog uticanja na implicitnu pretragu. Na primer, implicitni makro može odlučiti da ne odgovara određenoj implicitnoj pretrazi i pozvati *abort*, prekidajući ekspanziju makroa sa greškom, samim tim uklanjanjem iz liste implicitnih kandidata za ovu određenu implicitnu pretragu.

Ovo svojstvo je značajno komplikovalo implicitnu pretragu. Pored vođenja računa o oblasti važenja dostupnih implicitnih parametara, ugnježdenih implicitnih pretraga i povratka iz njih, uvođenje ekspanzije makroa u priču je suvišno. Zbog toga, počev od Skale 2.11, samo *Whitebox* makroi se proširuju tokom implicitne pretrage. *Blackbox* makroi učestvuju u implicitnoj pretrazi samo sa svojim potpisima, baš kao i regularni metodi, i njihova ekspanzija se dešava tek nakon što ih implicitna pretraga odabere.

4.1.7.3 Dejstvo na zaključivanje tipova

Kada primeni polimorfne metode nedostaju tipski argumenti, neovisno od toga da li je reč o regularnoj definiciji metode ili definiciji makroa, proveravač tipova pokušava da zaključi nedostajuće argumente koji se odnose na tipove. Tokom zaključivanja tipova, proveravač tipova skuplja ograničenja za nedostajuće tipove argumenata iz granica tipskih parametara, iz tipova termovskih argumenata, čak iz rezultata implicitnih pretraga (zaključivanje tipova radi zajedno sa implicitnom pretragom zato što Skala podržava analogon funkcionalnim zavisnostima). Na ova ograničenja se može gledati kao na sistem nejednakosti gde se nepoznati tipovi argumenata predstavljaju kao tipske promenljive i redosled je nametnut podtipskim relacijama.

Nakon skupljanja ograničenja, proveravač tipova počinje proces korak-po-korak koji, na svakom koraku pokušava da primeni izvesne transformacije nad nejednakostima, stvarajući ekvivalentan, za koji se očekuje da predstavlja jednostavniji sistem

nejednakosti. Cilj zaključivanja tipova je transformacija originalnih nejednakosti u jednakosti koje čine jedinstveno rešenje originalnog sistema.

U većini slučajeva, zaključivanje tipova je uspešno i kada je tako, nedostajući argumenti su zamenjuju tipovima predstavljenim rešenjem. Kako god, ponekad zaključivanje ne uspeva. Na primer, kada je tipski parametar T fantom, tj. neiskorišćen u termovskim parametrima metoda, njegov jedini ulaz u sistem nejednakosti će biti L <: T <: U, gde L i U predstavljaju donju i gornju granicu respektivno. Ako je L != U, tada ova nejednakost nema jedinstveno rešenje, i to dovodi do neuspeha u zaključivanju tipova.

Kad dođe do neuspeha u zaključivanju tipova, odnosno kada je nemoguće izvršiti dalju transformaciju, a tekuće stanje i dalje ima neke nejednakosti, proveravač tipova prekida zastoj. Prihvata sve do sada nezaključene tipske argumente, npr. one čije promenljive su i dalje određene nejednakostima, koje potom prisilno minimizuje, tj. izjednači sa njihvim donjim granicama. To dovodi do rezultata u kome su neki tipovi precizno zaključeni, a neki su zamenjeni sa prividno proizvoljnim tipovima. Na primer, tipski parametri bez ograničenja su zamenjeni tipom *Nothing*, koji je čest izvor zabune za početnike u Skali.

Od uvođenja def makroa, postavljeno je pitanje kako omogućiti korsnicima da prilagode izvođenje tipova svojim potrebama. Nakon mnogo nespešnih pokušaja, rešenje je pronađeno. Ako se izvođenje tipova za makro aplikaciju blokira, proveravač tipova prihvata tekući sistem nejednakosti i proizvodi parcijalno rešenje. To rešenje uključuje sve tipske argumente koji su zaključeni, kao i sintetisane tipove koji su tu umesto nezaključenih tipskih argumenata i koji su ograničeni prema odgovarajućim nejednakostima. Posle toga, proveravač tipova izvodi proširivanje makroa koristeći parcijalno rešenje kao tipske argumente makro aplikacije.

Makro aplikacije koje prođu kroz ovakvu vrstu tretmana od strane proveravača tipova ne mogu biti *Blackbox*, parcijalno zaključivanje tipova je omogućeno samo za *Whitebox* makroe. *Blackbox* makroi, nalik regularnim metodama, imaju nezaključene tipske argumente prisilno minimizovane kao što je prethodno opisano.

Iako izgleda egzotično, ovaj trik igra ključnu ulogu u materijalizaciji implicitnih parametara. U slučaju *Whitebox* implicitnih makroa korišćenjem ove tehnike, proveravač tipova može veoma precizno zaključiti tipove, omogućavajući napredno programiranje na nivou tipova.

4.1.8 Makro API-ji

Makro API-ji su učaureni u kontekstnom parametru implementacije makroa. U Skali 2.11, kontekst makroa može biti deklarisan sa *scala.reflect.macros.blackbox.Context* ili sa *scala.reflect.macros.whitebox.Context*. U Skali 2.10, postoji samo *scala.reflect .macros.Context* koji je ekvivalentan whitebox kontekstu. Razlika među tipovima konteksta se koristi da bi se utvrdila vrsta kojoj odgovara implementacija makroa. Postoji takođe i razlika u API-ju između konteksta *Blackbox* i konteksta *Whitebox* makroa, ali se sastoji od unutrašnjeg kompajlerskog mehanizma što je izvan oblasti ovog izlaganja.

Glavni deo makro API-ja dolazi iz *Context.universe* i *Context.mirror* koji predstavljaju *scala.reflect* univerzum i ogledalo koje odgovara tekućoj kompilacionoj jedinici. Kako god, postoje funkcionalnosti jedinstvene za makroe, koje će biti pokrivene u nastavku sekcije.

Prvo, tu su *Context.macroApplication* koji obuhvata proširenu aplikaciju makroa i *Context.prefix* koji sadrži prefiks makro aplikacije. Oni postoje da bi se pristupilo argumentima makro apilikacije preko parametara implementacije makroa.

Drugo, konteksti podržavaju emitovanje dijagnostičkih poruka. Moguće je prevremeno završavanje makro aplikacije sa prilagođenom greškom putem *Contex. abort*. Takvi API-ji zahtevaju poziciju, td kompajler može da veže poruku sa lokacijom u kodu. Ovo je moćan način proizvodnje domenski specifičnih poruka o greškama.

Treće, moguće je pristupiti unutrašnjem mehanizmu kompajlera. Makro API-ji su se postupno razvili u naprednu funkcionalnost, primer su rutine niskog nivoa za upravljanje tabelama koje se mogu koristiti za rad sa ograničenjima *scala.reflect* jezičkog modela.

4.1.9 Definicije makroa naspram ostalih metoda

Meta podaci Većina definicija u Skali, isključujući samo pakete i paketske objekte, dopuštaju korisnički definisane anotacije koje se mogu čitati u vreme kompilacije i/ili u vreme izvršavanje. Definicije makroa nisu izuzetak. Zahvaljujući svojoj sposobnosti da definicijama prikupe meta podatke iz vremena kompjaliranja, anotacije su se koristile da se dele informacije koje uključuju uzajamno dejstvo između makroa.

Modularnost Skala poseduje bogat skup svojstava za upravljanje oblašću važenja i pristupom njenim definicijama. Sva ta svojstva važe i slučaju definicija makroa. Kao i sa regularnim definicijama, moguće je imati definicije makroa kako u lokalnoj

oblasti važenja tako i u pridruženom metodu zagrađujuće definicije. Ako je neophodno definicije makroa mogu biti privatne i zaštićene (eng. protected).

Nasleđivanje Jedna od ključnih karakteristika regularnih definicija je dinamičko raspoređivanje. Skala podržava uobičajenu mešavinu svojstava koja podrazumeva dinamičko raspoređivanje: podklase, prezapisivanje, apstraktnost i finalnost. Def makroi se proširuju u vreme kompilacije, što znači da je dinamičko raspoređivanje izvan domašaja, kao i neka svojstva koja se na njega odnose. Posebno, to znači da definicije makroa ne mogu biti apstraktne i ne mogu prezapisivati apstraktne metode.

Impliciti Zavisno od potpisa, regularne definicje implicita mogu služiti kao podrazumevane vrednosti implicitnih parametara ili kao konverzija između inače nekompatibilnih tipova. Obe ove uloge implicitnih definicija mogu imati koristi od metaprogramiranja u vreme kompajliranja, pa je dopušteno da definicije makroa budu impliciti takođe. Tehnika materijalizacije implicita omogućena implicitnim makroima predstavlja jednu od najvažnijih upotreba definicija makroa.

Konstruktori U vreme pisanje, nemoguće je definisati konstruktor kao defincju makroa.

Signature Definicije makroa imaju iste elemente u potpisu kao i regularne definicije - ime, opcioni tipski parametri, opcioni termovski parametri, opcioni povratni tip. Nema teorijskih ograničenja za parametre definicija makroa, ali se u praksi ne dopuštaju definicije makroa koje imaju podrazumevane parametre. Postoji ograničenje u zaključivanju tipova. U slučaju regularnih definicija se povratni tip može zaključiti iz tela metoda. Kako definicije makroa imaju neuobičajeno telo, standardni algoritam više nije primenljiv za zaključivanje povratnog tipa. U inicijalnoj verziji definicija makroa koje su zahtevale da implementacije makroa prihvataju i vraćaju izraze, zaključivanje povratnog tipa definicije makroa se vršilo pomoću povratnog tipa odgovarajuće implementacije. Na primer, ako implementacija makroa vraća $Expr[Parser[Array[Int]]]$ onda se povratni tip definicije može zaključiti kao $Parser[Array[Int]$. Sada kada se dopušta da implementacije makroa vraćaju obična stabla, zaključivanje povratnog tipa definicije makroa nije moguće. Ovo svojstvo je označeno kao zastarelo, i nije u planu njegova podrška u sistemu za rad sa makroima.

Preopterećenje Kao i regularne definicije, definicije makroa se mogu preopteretiti - i regularnim definicijama i drugim definicijama makroa. To ne stvara probleme, jer se razrešenje prezasićenja dešava pre proširivanja makroa. Dodatno, pošto se ne prikazuje bajtkod definicija makroa (zbog toga što definicije postoje jedino za vreme kompilacije), ne postoji ograničenje da definicije makroa moraju imati drugačiju signaturu u odnosu na obrisane signature drugih metoda.

Tipovi Kada je reč o ugnježdavanju, Skala nema mnogo ograničenja. Termovi mogu biti ugnježdeni u tipove, tipovi mogu biti ugnježdeni u termove, slično važi i za definicije. Kao rezultat, neki napredni tipovi, tačnije složeni i egzistencijani tipovi, mogu sadržati definicije. Dok egzistencijalni tipovi mogu samo definisati apstraktne *val* promenljive i apstraktne tipove, složeni tipovi mogu uključivati bilo koje apstraktne članove.

Iz izložene diskusije, može se zaključiti da su definicije makroa relativno dosledno proširenje običnih regularnih definicija. Razlike su: 1) ograničenje na prezapisivanje, jer definicije makroa ne postoje u vreme izvršavanja, 2) skoro nepostojeće zaključivanje tipova, zbog nesvakidašnjih tela definicija makroa, 3) nemogućnost da budu sekundarni konstruktori ili da imaju podrazumevane vrednosti.

4.1.10 Makro aplikacije

Ista analiza koja je primenjena na defincije makroa, može se primeniti i na makro aplikacije. Ako se prođe kroz sve primene metoda i kontekste gde se primenjuju makroi, pokušaji da se korišćenje regularnih metoda zameni sa korišćenjem makroa, daju sledeće rezultate:

Potpuno specificirane aplikacije Ako aplikacija makroa ima sve tipske argumente i sve termovske argumente specificirane prema potpisu definicije makroa, tada je kompajler proširuje tako što poziva odgovarajuću implementaciju makroa sa ovim argumentima (ceo postupak je opisan u sekciji 4.1.7).

Nedostajući tipski argumenti Kada makro aplikacija nema tipske argumente, a odgovarajuća definicija makroa ih ima, zaključivanje se dešava pre ekspanzije makroa.

Parcijalne aplikacije Regularne definicije mogu biti primenjene parcijalno, npr. njihove primene mogu imati manje termovskih argumenata nego odgovarajuće definicije. Definicije makroa takođe podržavaju parcijalnu aplikaciju, ali sa izvesnim ograničenjima. Liste sa nedostajućim implicitnim argumentima su objašnjene dole. Drugi slučajevi parcijalne aplikacije uključuju nedostajuće argumente koji ne mogu biti zaključeni. Za regularne definicije, ovo je rešeno eta ekspanzijom koja pretvara parcijalne aplikacije u objekte funkcije kojima se preostali argumenti mogu proslediti u vreme izvršavanja. Kako god, definicije makroa se proširuju u vreme kompilacije, pa eta ekspanzija za njih nije dopuštena.

Nedostajući implicitni argumenti Pre ekspanzije makroa, kompajler će se postarati da aplikacija makroa razreši svoje implicitne argumente. Ako implicitni

argumenti nisu specificirani eksplicitno, proveravač tipova započinje implicitnu pretragu da bi ih zaključio. Nema ograničenja na poreklo argumenta - regularne *val* promenljive, regularne definicije i definicije makroa su dopuštene.

Nedostajući podrazumevani argumenti Za razliku od regularnih definicija, definicije makroa ne mogu imati podrazumevane vrednosti, pa tako ni njihove primene.

Promenljivi argumenti Makro aplikacije mogu imati nula ili više argumenata koji odgovaraju istom promenljivom argumentu definicije makroa. U tom slučaju, prema podsekciji 4.1.4 i implementacije makroa moraju imati promenljivi parametar. Sistem za rad sa makroima obmotava svaki promenljivi argument ponaosob, potom prosleđuje kolekciju ovih argumenata u promenljivi parametar implementacije makroa.

Uklanjanje sintaksnih ulepšavanja Zanimljiva osobina Skale je da je značajan broj njenih jezičkih svojstava, npr. dodele, poklapanje šablona, apstrakcije niza, inperpolacije niski itd, nakon uklanjanja sintaksnih ulepšavanja pretvoren u primene metoda. Kao rezultat, ova svojstva transparento mogu biti dostupna makroima.

Kao što je i izloženo, aplikacije makroa gotovo savršeno se integrišu sa postojećom infrastrukturom primena metoda. Razlike su 1) specijalan tretman $Whitebox$ makroa od strane proveravača tipova, jer ekspanzija makroa može upravljati zaključivanjem tipova, 2) skoro nepostojeća parcijalna aplikacija, jer se makro aplikacije proširuju u vreme kompilacije, 3) nedostatak podrške za podrazumevane i imenovane argumente, jer inicijalna verzija nije podržavala ove funkcionalnosti.

4.2 Zaključak poglavlja o makroima

Definicije makroa su načinile preokret u metaprogramiranju u Skali. Pre uvođenja makroa u Skalu 2.10, metaprogramiranje u vreme kompajliranja je bilo ograničeno i zastarelo programiranje na nivou tipova, u nekim slučajevima obimno, ali nestablino i teško za distribuiranje kompajlerskim priključcima. Definicije makroa su prevazišle oba ova problema, pružajući moćan API za refleksiju i način da se transparentno pakuju i distribuiraju metaprogrami iz vremena kompajliranja.

Definicije makroa duguju veliki deo svog uspeha sličnosti sa regularnim metodama. Izvan konteksta makroa, mnoga postojeća Skalina svojstva uklanjanjem sintaksnih ulepšavanja su svedena na pozive metoda - ili na pozive metoda sa specijalnim imenima kao što su *apply* ili *unapply*, ili na metode sa specijalnim značenjima poput implicita. Definicije makroa su omogućile da se zadrži sličan korisnički intefejs

i semantika za postojeća Skalina svojstva, uporedo generišući kod i obezbeđujući programiranje u vreme kompajliranja.

Još jedan bitan faktor koji je doprineo popularnosti makroa je lako distribuiranje. Većina funkcionalnosti koje su omogućavali makroi je bila dostupna kompajlerskim priključcima, iako na opširan način. Omogućavanje kompajlerskog priključka zahteva prilagođeni kompajlerski fleg, dok omogućavanje definicije makroa ne zahteva ikakvo ručno konfigurisanje. Rezultat toga je da korisnici makroa ne moraju ni da znaju da prilagođavaju svoj kompajler, što predstavlja ključni napredak u korisničkom iskustvu.

Scala.reflect je odigrala ključnu ulogu u omogućavanju definicija makroa. Sa jedne strane ova biblioteka je pružila bogat metaprogramerski *API*, koji inače zahteva značajno vreme za dizajn i implementaciju. Ironično, većina zamerki na račun makroa je posledica komplikovanog *API*-ja kao i čudnih kodnih idioma *scala.reflect*-a.

5 Primer primene makroa

U ovom poglavlju će na primeru parser kombinatora biti prikazano kako se pomoću makroa dostižu značajna ubrzanja u odnosu na kombinatore iz Skalinog paketa *scala.util.parsing.combinator*. Ovaj paket je primer unutrašnjeg DSL-a, koji se sastoji od biblioteke parser kombinatora - funkcija i operacija definisanih u Skali koje služe kao gradivni blokovi za parsere.

Parser kombinatori i njihova implementacija su popularni u funkcionalnom programiranju. Njih je inicijalno Wadler [14] uveo radi predstavljanja monada. Od tada su ugrađeni u programske jezike kao biblioteke, kao što je *Parsec* u Haskelu i biblioteka u Skali. Ove biblioteke su fokusirane na proizvodnju jednog rezultata. Takve biblioteke za kombinatore su proširene da mogu da rade sa većom klasom gramatika.

Naspram parser kombinatora nalaze se parser generatori, primer su *Yacc*, *Antlr*, *Happy*. Iako ovi alati imaju dobre performanse, ne podržavaju kontekstnu osetljivost koja se zahteva u parsiranju protokola korišćenih od strane mrežnih aplikacija [13].

Pristup zasnovan na makroima povezuje oba sveta u smislu svojstava parsera: lak za korišćenje, kontekstno osetljiv, kompozabilnost, specijalizovanost i performanse.

5.1 Parser kombinatori

Parser kombinatori omogućavaju da se na intuitivan način pišu parseri (današnja upotreba parsera se najviše odnosi na struktuirane podatke). U funkcionalnim jezicima, poput Skale, oni su implementirani kao funkcije višeg reda koje preslikavaju ulaz u njegovu struktuiranu reprezentaciju. Parseri pisani na takav način blisko oslikavaju formalni opis gramatike. Dodatno, kako su ugrađeni u objektni jezik, oni su modularni, kompozitni i spremni za izvršavanje.

Glavni razlog zašto nisu ušli u širu upotrebu je što nisu efikasni. Apstrakcije koje omogućuju izražajnost su vremenski veoma zahtevne. Uprkos njihovom deklarativnom zapisu, opis gramatike je isprepletan sa procesiranjem ulaza, pa se tako tokom obrade ulaza delovi gramatike izgrađuju iznova više puta.

Kompozicija parsera je uglavnom statična. Struktura parsera je potpuno poznata pre njegovog pokretanja za dati ulaz.

Razdvajanjem kompozicije od procesiranja ulaza, može se eliminisati vremensko preopterećenje, i time dobiti efikasan parser koji će se jednostavno izvršavati nad

ulazom. Drugim rečima, moguće je parser kombinator pretvoriti u generator parsera u vreme kompilacije.

U Skalinom okruženju, postoje dva glavna pristupa optimizacijma u vreme kompajliranja. Tradicionalni način je da se implementiraju kompajlerski priključci (eng. *plugin*). Najveći nedostatak ovakvog pristupa je što u potpunosti izlaže mehanizam kompajlera. Programer mora da bude dobro upućen u Skalina kompajlerska stabla, koja su uopštenija od produkcionih pravila opisa gramatike. Alternativni pristup je da se koriste tehnike metaprogramiranja, kao što su makroi. Takve tehnike dopuštaju da se operiše sa opisima parser programa koji su na višem nivou, čineći implementaciju lakšom i pogodnijom za proširivanje.

5.2 Implementacija kombinatora makroima

Iz ugla korsnika, parser se može pisati slično kao i slučaju korišćenja standardne biblioteke za parser kombinatore. Ali uz pomoć makroa se razdvaja statička kompozicija parser kombinatora od dinamičkog procesiranja ulaza u vreme kompajliranja. To je postignuto u dvofaznoj transformaciji. Parser je pisan unutar $FastParser$ konteksta, makroa koji čini celinu unutar koje postoji parser.

Tokom prve faze se analizira struktura parsera i produkciona pravila se svode na pravila u jednoj liniji, tako da parser bude uprošćen i da sadrži lanac elementarnih kombinatora. Tokom koraka dovođenja pravila u jednu liniju, razmatraju se rekurzivni parseri i pozivi parsera u različitim $FastParser$ kontekstima. U drugoj fazi, proširuju se definicije elementarnih kombinatora, korišćenjem kvazinavoda.

Ova transofmacija nije trivijalna za kombinatore višeg reda, kao što je $flatMap$. Analizira se telo funkcije prosleđeno kombinatoru i proširuje se ako je neophodno. Moguća su i pravila koja prihvataju parametre.

5.3 Definicije parser kombinatora

Parser kombinatori su funkcije koje preslikavaju ulaz u rezultat parsiranja koji može biti uspeh ili neuspeh. U sledećem primeru prikazana je implementacija parser kombinatora u Skali. Crta $Parsers$ se ponaša kao kontekst u kom su implementirani parseri. Apstrakcija nad tipom elemenata ulaza je $Elem$, a nad tipom ulaza je $Input$. $Input$ je tipski alijas za $Reader$, koji predstavlja interfejs za pristup tokenima u nizu ili toku. Uprošćeni $Reader$ se može definisati nad nizom karaktera ili nad niskom. Osnovni tip elemenata u ovom slučaju je $Char$.

```
1  trait Parsers {
2    type Elem
3    type Input = Reader[Elem]
4    abstract class ParseResult[T]
5    case class Success(res: T, next: Input)
6      extends ParseResult[T] {
7    def isEmpty = false
8    }
9    case class Failure(next: Input) extends ParseResult[T] {
10   def isEmpty = true
11 }
12 abstract class Parser[T]
13   extends (Input => ParseResult[T]) {
14     def | (that: Parser[T]) = Parser[T] { pos =>
15     val tmp = this(pos)
16     if(tmp.isEmpty) that(pos)
17     else tmp
18 }
19 def flatMap[U](f: T => Parser[U]) = Parser[U] { pos =>
20   val tmp = this(pos)
21   if(tmp.isEmpty) Failure(pos)
22   else f(tmp.res)(tmp.next)
23 }
24 def map[U](f: T => U) = Parser[U] { pos =>
25   val tmp = this(pos)
26   if(tmp.isEmpty) tmp
27   else Success(f(tmp.res), tmp.next)
28 }
29 def ~[U](that: Parser[U]) : Parser[(T,U)] =
30   for(r1 <- this;r2 <- that) yield (r1, r2)
31 }
32 def Parser[T](f: Input => ParseResult[T]) =
33   new Parser[T] {
34   def apply(pos: Input) = f(pos)
35 }
36 }
37   abstract class Reader[T] {
38     def first: T
39     def next: Reader[T]
40     def atEnd: Boolean
41   }
```

Jedan od bitnijih kombinatora je *flatMap*, koji vezuje rezultat parsiranja za parser. Ovo je monadički operator vezivanja za parser kombinatore, dopušta da se donose odluke bazirane na rezultatu prethodnog parsera. Alternirajući kombinator '|' parsira desnu stranu parsera samo pod uslovom da leva strana završi parsiranje sa neuspehom. *Map* kombinator transformiše vrednosti rezultata parsiranja. Finalno, kombinator ~ izvršava sekvencionisanje, gde je od značaja i rezultat desne i leve strane parsera.

U skupu elementarnih kombinatora dodatno se nalaze i kombinatori:

- **lhs ~> rhs** završava sa uspehom ako se to dogodi u slučaju obe strane lhs i rhs, ali je od interesa rezultat parsiranja desnog parsera rhs
- **lhs <~rhs** završava sa uspehom ako se to dogodi u slučaju obe strane lhs i rhs, ali je od interesa rezultat parsiranja levog parsera
- **rep(p)** ponavlja korišćenje parsera p za parsiranje ulaza sve dok p ne vrati grešku. Rezultat je lista uzastopnih rezultata parsera p.
- **repN(n,p)** koristi p tačno n puta za parsiranje ulaza. Rezultat je lista n uzastopnih rezultata parsera p.
- **repsep(p,q)** ponavlja korišćenje parsera p, naizmenično sa parserom q. Parser p vrši parsiranje sve dok ne dođe do neuspeha, a u tom trenutku ga zamenjuje q ... Parsiranje se prekida kada ni parser p, ni parser q ne mogu da se uspešno isparsiraju tekući ulaz. Rezultat je lista rezultata parsera p. Npr, repsep(term, ",") parsira listu termova razdvojenu zarezima, a kao rezultat vraća listu termova.

Funkcionalna implementacija prikazana u prethodnom primeru dovodi do loših performansi, jer:

- Kako je svaki parser funkcija, a funkcije su objekti u Skali, primena funkcija dovodi do poziva metoda. Kompozitni parser je sačinjen od mnogo manjih parsera, a kada se primeni nad ulazom, pored toga što konstruiše parser pri svakoj primeni, pokreće lanac metoda. Takav način rada dovodi do dugog izvršavanja, nastalog tokom obrade velikog broja metoda. Upotreba funkcija višeg reda dodatno uvećava taj vremenski trošak.
- Konstruišu se mnogi posredni rezultati parsiranja tokom izvršavanja parsera: za svaki kombinator se prosleđuju do tada isparsirani delovi, plus pozicija do koje se stiglo u parsiranju ulaza koji su upakovani u *ParseResult* objekat.

Kao zaključak, upravo su mehanizmi apstrahovanja jezika koji omogućuju komponovanje kombinatora glavni razlog usporavanja njihovog rada.

5.4 Kombinatori zasnovani na makroima

Sledi opis implementacije biblioteke za parser kombinatore zasnovane na makroima. U primeru ispod je naveden primer korišćenja ove biblioteke. Nasleđuju se funkcionalnosti *MyParser* objekta, što omogućava pristup kombinatorima kao i oblasti važenja *FastParser*. Potom se deklariše parser u oblasti važenja *FastParser*-a. Ovaj parser je veoma sličan standardnoj implementaciji parser kombinatora[12].

I sledećem primeru je naveden parser izgrađen od parser kombinatora za parsiranje JSON objekata.

```
1  import MyParsers._
2
3  val jsonParser = FastParser {
4    def value: Parser[Any] = obj | arr | stringLit |
5    decimalNumber | "null" | "true" | "false"
6    def obj: Parser[Any] = "{" ~> repsep(member, ",") <~ "}"
7    def arr: Parser[Any] = "[" ~> repsep(value, ",") <~ "]"
8    def member: Parser[Any] = stringLit ~ (":" ~> value)
9  }
```

Kako za JSON objekat važi da je ili:

• primitivna vrednost, poput decimalnog boja, string literala, bulovske ili null vrednosti.

• ili niz vrednosti (niz funkcija).

• ili asocijativna tabela ključ-vrednost parova (objekat funkcija)

može se pozvati *value* produkciono pravilo na sledeći način:

```
1  val cnt = "{\"firstName\": \"Jelena\"," +"\"age\": 25}"
2  jsonParser.value(cnt) match {
3    case Success(result) => println("uspeh :" + result)
4    case Failure(error) =>println("nesupeh : " + error)
5  }
```

Kao što je već spomenuto razdvojenost između spoljašnjeg i unutrašnjeg makro sveta je paradigma na kojoj se zasniva arhitektura biblioteke za rad sa makroima. Stoga postoji razlika između onog što korisnik biblioteke vidi, tzv. intefejs i makro svet gde su kombinatori optimizovani, tzv. implementacija. Primer

ispod oslikava ovo razdvajanje. Objekat *MyParsers* je ulazna tačka ka interfejsu. Primese koji se ovde spominju su *BaseParsers* i *TokenParsers*, pošto se radi sa ulazima koji su niske, izgrađene od elemenata tipa *Char*.

Svaka crta primesa iz interfejsa obezbeđuje pristup raznim kombinatorima. Crta *BaseParsers*, npr definiše kombinator sekvencionisanja ~. Anotacija @*compileTimeOnly* obezbeđuje da korisnik dobije poruku o grešci u vreme kompajliranja ako se kombinator ne koristi unutar *FastParser* konteksta i da se greska ne izbacuje. Oblast važenja *FastParsers*-a je mesto gde se odigrava cela radnja. U pitanju je *Whitebox* makro koji prihvata skup pravila, koja potom transfomiše i optimizuje, a kao rezultat vraća objekat koji je podtip klase *FinalFastParser*. Ovaj objekat sadrži optimizovane implementacije za svako pravilo definisano u *FastParser*-u. Korišćenje *Whitebox* makroa obezbeđuje prerađivanje tipova ovog objekta, tako da se pravila mogu pozivati spolja. Da se koristio *Blackbox* makro, pozivanje pravila *value* iz *jsonParser*-a u prethodnom primeru bi rezultovalo greškom u vreme kompajliranja. U implementacionom sloju, *MyParsersImpl* klasa takođe meša funkcionalnosti za transformisanje parsera i proširivanje kombinatora:

• crte *RulesTransformer* i *RulesInliner* sadrže implementacije za preprocesiranje pravila

• crte *BaseParsersImpl*, *TokenParsersImpl*, *RepParsersImpl* i *FlatMapImpl* sadrže funkcionalnosti za ekspanziju pravila. One nasleđuju *ParserImplBase* crtu koja definiše funkciju za ekspanziju.

• crta *RuleCombiner* kombinuje pravila definisana u makrou *FastParser* u finalni objekat.

• crta *StringInput* je indikator da se radi sa ulazima koju su tipa niske.

• crta *IgnoreParseError* pokazuje da se ignoriše bilo koji oblik prikazivanja grešaka.

```
1
2  // interfejs
3  object MyParsers extends BaseParsers[Char, String]
4    with TokenParsers with RepParsers ... {
5      def FastParser(rules: => Unit): FinalFastParser =
6        macro MyParsersImpl.FastParser
7  }
8
9
10 trait BaseParsers[Elem, Input] {
11
```

```
implicit class BaseParserHelpers[T](p1: Parser[T]) {
  $@compileTimeOnly$
  def ~[U](p2: Parser[U]): Parser[(T, U)] =
    throw new NotImplementedError
}

trait TokenParsers { ... }
//   implementacija
class MyParsersImpl(val c: Context) extends BaseParsersImpl
with TokenParsersImpl with RepParsersImpl
with RulesTransformer with RulesInliner
with FlatMapImpl with RuleCombiner
with StringInput with IgnoreParseError {
  def FastParser(rules: c.Tree): FinalFastParser = ...
}
trait ParserImplBase {
  def expand(tree: c.Tree, rs: ResultsStruct ): c.Tree = ...
}
trait BaseParsersImpl extends ParserImplBase { ... }
trait TokenParsersImpl extends ParserImplBase { ... }

}
```

U daljem tekstu će biti prikazano kako se optimizuju pravila unutar *FastParser* makroa.

5.4.1 Transformacija pravila

U prethodnom primeru su navedene *dummy* deklaracije kombinatora. Cilj da se ti kombinatori implementiraju u makro svetu, korišćenjem kvazinavoda. Jednostavna, lokalna zamena kombinatora sa efikasnijim kodom nije dovoljna. Razmatranjem parsera $parser1$ u primeru ispod, u slučaju pravila $rule2$, ono se jednostavno proširuje u kombinator sekvencionisanja, ali i dalje ostaje poziv pravila $rule1$. Tokom izvršavanja parsera, ovo će rezultovati usporenjem nastalim usled poziva funkcije. Poželjno je da se i pravilo $rule1$ proširi ovde.

Kako bi se dopustile globalnije optimizacije za kombinatore, pre koraka ekspanzije makroa prvo se vrši korak preprocesiranja. Ovaj korak se sastoji od prolaska kroz svako pravilo po dubini. Kadgod se naiđe na pravilo, u liniju se unese njegova desna strana. Nakon ove faze, pravilo $rule2$ se prezapisuje sa $rule2 =' d'^{\sim}('a'^{\sim}'b')$.

Posebnu pažnju bi trebalo obratiti na rekurzivne parsere. U njihovom slučaju se koristi klasična tehnika praćenja pravila koja je prikazana do sada. Odluka da li

će se vršti svođenje pravila u liniju se zasniva na tome da li se rekurzivni poziv pojavljivao ranije ili ne. Uzajamno rekurzivni parseri se obrađuju na sličan način, dok se izvodi korak preprocesiranja za svako pravilo. Odatle, pravila *rule*3 i *rule*4 iz sledećeg primera, s obzirom na to da su uzajamno rekurzivna se ne transfomrišu.

```
val parser1 = FastParser {
  def rule1 = a ~ b
  def rule2 = d ~ rule1
  def rule3 = y ~ rule4
  def rule4 = rule3 | x
}

val parser2 = FastParser {
  def rule1 = c ~ parser1.rule2
}
```

5.4.1.1 Eksterni pozivi

Pravilo se može pozivati izvan *FastParser* konteksta u kome je definisano. Ova funckionalnost je veoma korisna u slučaju da se optimizovani parser iznova korisiti na različite načine u različitim bibliotekama. U prethodnom primeru, pravilo *rule*1 u parseru *parser*2 poziva pravilo *rule*2 definisano u *parser*1 zbog čega se pravilu *rule*2 dodaje prefiks *parser*1. Tokom faze preprocesiranja, takvi eksterni pozivi se uzimaju u obzir. Kadgod se naiđe na kod oblika $parser.rule(args)$, događa se sledeće:

• Proverava se da li je parser podtip od *FinalFastParser*, jer će *FastParser* makro proširiti *parser*1 u instancu ove klase.

•Potom se proverava da li je pravilo definisano nad tim objektom i da li su mu prosleđeni odgovarajući argumenti. Štaviše, dobija se i njegovo AST koje je sadržano u ~saveAST anotaciji.

• Kada su oba prethodno navedena uslova zadovoljena, može se izvršiti svođenje pravila u jednu liniju. U primeru iznad, *parser*2.*rule* će se transformisati u:

```
def rule1 = c ~ (d ~ (a ~ b))
```

Kada se poziv eksternog pravila zamenjuje AST-om, pažljivo treba stavljati prefikse kako bi se odgovarajuće pravilo pozivalo: parser1.rule1 je drugačije u odnosu na parser2.rule1.Tako da se desna strana 'd'~rule1 zapisuje kao 'd'~parser1.

rule1 prvo. Takođe parser1 mora biti proširen pre parser2, jer je potrebno pristupiti realnom tipu kako bi se pozivala pravila, inače bi parser1 video samo objekat tipa $FinalFastParser$ koji ne sadrži ikakve metode. Kako je $FastParser$ $Whitebox$ makro, njegov realni tip se saznaje tek nakon ekspanzije makroa. Prema tome ni Skalin kompajler ne bi dopustio kompajliranje koda gde parser1 poziva pravilo definisano u parser2. To je takođe jedan od razloga zašto uzajamno rekurzivni pozivi između parsera iz različitih $FastParser$ konteksta nisu dopušteni.

5.4.2 Prezapisivanje pravila

Nakon preprocesiranja i svođenja pravila u liniju, sledeći korak je proširivanje primitivnih kombinatora. Glavni deo prezapisivanja je sadržan u funkciji *expand* koja je navedena u primeru implementacije $MyParsersImpl$. Ova funkcija prihvata kao argument drvo koda koji treba da se proširi, kao i $ResultStruct$, koji se bavi rezultatima parsiranja. Cilj je minimizovati kreiranje varijabli po svakom rezultatu parsiranja, i da rezultat sadrži indikator uspeha, odnosno neuspeha parsiranja.

Tokom proširivanja implementacije kombinatora, promenljiva zadužena za uspeh (koja je globalna promenljiva) treba da bude postavljena na $true$ ili $false$ zasnovano na uspehu trenutnog parsera. U primeru ispod je navedena implementacija *expand* funkcije za kombinator sekvencionisanja. Pritom se koriste kvazinavodi da bi se prepoznao ovaj kombinator. Rekurzivno se proširuje leva strana lhs. Ako je rezultat uspeh, razmatra se dalje rekurzivno proširena desna strana rhs. U slučaju greške, parser prekida izvršavanje, i propagira grešku do vrha.

```
1  trait BaseParsersImpl extends ParserImplBase {
2  override def expand(tree: c.Tree, rs: ResultsStruct):
3  c.Tree = tree match {
4  case q"$lhs ~[$_] $rhs" => q"""
5  ${expand(lhs, rs)}
6  if (success) {
7  ${expand(rhs, rs)}
8  }
9  """
10 case q"$lhs | [$_] $rhs" => ...
11 ...
12 }
13 ...
14 }
```

Za parser koji parsira karakter ‘a’ koji prati karakter ‘b’ (‘a’ ’~‘b’), dobija se prošireni kod:

```
var success = false
var result1 = "
var result2 = "
...
if (inputpos < inputsize && input(inputpos) == a){
  result1 = a
  inputpos += 1
  success = true
}
else {
  success = false
  error = "expected a at " + inputpos
}
if (success){
  if (inputpos < inputsize && input(inputpos) == b){
    result2 = b
    inputpos += 1
  success = true
  }
  else {
    success = false
    error = "expected b at " + inputpos
  }
}
```

Prirodno, moguće je da korisnik definiše sopstvena proširivanja, ako želi da doda neke specifične optimizovane verzije kombinatora. Sve što je neophodno je da nasledi crtu *ParserImplBase* i da pregazi (eng. override) *expand* funkciju.

5.4.2.1 Upravljanje rezultatima parsiranja

Kao što se može uočiti u primeru iznad, svaki privremeni rezultat se čuva u novoj promenjivoj. Sa kombinatorom sekvencionisanja, generišu se dve promenljive za rezultat, jedna za levu, druga za desnu stranu; ali nije potrebno konstruisati rezultujući par sve do njegove upotrebe. Prate se zavisnosti tokom ekspanzije makroa u instanci *ResultStruct* klase. Ova klasa sadrži metode za stvaranje, kombinovanje, dodeljivanje i praćenje varijabli za rezultate parsiranja. Kako kombinator sekvencionisanja ne zahteva eksplicitno generisanje promenljivih (to je završeno u rekurzivnim proširivanjima leve i desne strane), u nastavku je prikazan primer upotrebe *ResultStruct* klase za proširivanje *map* kombinatora. Treba napraviti novu *ResultStruct* instancu pre proširivanja leve strane a. Ako

je parsiranje uspešno, kombinuje se rezultatom do kog se do tada došlo korišćenjem *combine* metoda. Potom se taj rezultat dodeljuje novoj promenjivoj $assignNew$. U slučaju da je **a** nastao sekvencionisanjem drugih kombinatora, rezultujuća n-torka bi se generisala ne mestu poziva *combine* funkcije.

5.4.3 Sastavljanje parsera

Do sada, prošireno je svako pravilo definisano u oblasti važenja $FastParser$. Suštinski, iz opisa parsera na nivou interfejsa, generisan je efikasan parser korišćenjem metode rekurzivnog spusta. Finalni korak uključuje pakovanje ovih pravila u objekat, tako da se kod može pozivati iz spoljašnjeg sveta, kao što je navedeno u primeru JSON parsera na početku poglavlja. Kao što je već spominjano, korišćene su pogodnosti *whitebox* Skalinih makroa da bi se genrisao podtip od $FinalFastParser$. U ovom objektu, generiše se metod za svako pravilo na nivou interfejsa. Za svako pravilo oblika def $rule(args : ...)$ generiše se istoimeni metod, kome se dodaju dva dodatna parametra: def $rule(in : Input, args : ..., offset : Int) : ParseResult[T]$. In parametar predstavlja ulaz na kome će se primenjivati parser, a $offset$ je pozicija u ulazu od koje počinje parsiranje. Povratni tip je $ParseResult[T]$, gfe je T povratni tip originalnog pravila.

U daljem tekstu su prikazani načini rukovanja sa netrivijalnim kombinatorima kao što su $flatMap$ i pravila sa dodatnim parametrima.

5.4.3.1 Kombinator flatMap

Kao što je do sada navedeno, $flatMap$ kombinator proizvodi novi parser pomoću funkcije koja preslikava rezultat u parser. Dopušta kontekstno osetljive parsere kao što je parser koji čita ceo broj n, praćen sa n karaktera.

```
1 FastParsers {
2   def rule = number flatMap { n => take(n) }
3 }
```

Primenjujući prethodno navedeno pravilo na ulaz "5abcdefg", dobiće se rezultat "abcde". Moguće je primenjivati transformacije na funkciju prosleđenu kao argument $flatMap$ kombinatoru, čija signatura je prikazana u sledećem primeru:

```
def flatMap[U](f: T => Parser[U]): Parser[U]
```

Funkcija f je ograničena na to da bude ili anonimna lambda funkcija ili parcijalna funkcija. U oba slučaja, poslednji iskaz u lambda izrazu ili poslednji iskazi svakog slučaja u parcijalnoj funkciji su tipa $Parser[U]$. Korišćenjem te činjenice, može se proširiti upotreba ovog kombinatora. Kad god se naiđe na šablon oblika

```
a.flatMap{ params => body; ret }
```

dešava se sledeće: • Proširuje se a, rekurzivnim pozivanjem funkcije *expand* nad njim

• Proširuje se *ret*. Ako je jednostavna lambda funkcija, ona se proširi. Inače, u pitanju je parcijalna funckija, pa se proširuje rekurzivno poslednji izraz svakog slučaja. Kreira se nova funkcija oblika:

```
{ params => body; expand(ret) }
```

• Na kraju se primeni rezultat od a na rezultujuću funkciju.

5.4.3.2 Pravila sa parametrima

Pravila definisana u spoljašnjem svetu mogu imati parametre, jer je reč o Skalinim metodama, koje prihvataju dodatne argumente. U primeru ispod su prikazana takva pravila. Kada se pozivaju pravila sa parametrima, parametri su konkretni ili specificirani. U sledećem primeru, pravilo *rule*2 poziva pravilo *rule*1 sa specifičnim parametrom.

```
FastParser {
//pravila sa parametrima
def rule1(x: Int) = repN(x, b)
def rule2 = a ~ rule1(5)
//uzajamno rekurzivna pravila sa parametrima
def rule3(p: Parser[List[Char]], y: Int): Parser[Any]
= a ~ p ~ rule4(y)
def rule4(x: Int): Parser[Any]
= rule3(repN(x, c), x + 1) | b
}
}
```

Odatle, tokom faze preprocesiranja prosleđuju se konkretni argumetni kombinatorima na mestu upotrebe. Kao rezultat pravilo *rule*2 se transformiše u:

```
def rule2 = a ~ repN(5, a)
```

Rekurzivna pravila su komplikovanija. Pre svega ne možemo direktno prosleđivati primitivne kombinatore poput $repN(x,'c')$, jer je tip njihovog interfejsa $Parser[T]$, a prošireni kod ne može sadržati ovaj tip. Rešenje je da se prvo pretvore pravila koja imaju parametre tipa *Parser* u pravila koja uzimaju vrednosti ekvivalentnog proširenog tipa. Signatura pravila *rule*3 postaje:

```
def rule3(in: InputType,
p: (InputType, Int) => ParseResult[List[Char]],
y: Int,
offset: Int = 0): ParseResult[Any]
= ...
```

Za svaku lokaciju gde se poziva *rule*3 sa specifičnim kombinatorom, kreira se proširena i optimizovana funkcija za ovaj kombinator. Za tekući primer, treba kreirati funkciju koja sadrži ekvivalent za *repN*:

```
def anonymous$1$(in: InputType, x: Int, offs: Int)
= prosireni kod za repN(x, c)

// Finalno, poziva se modifikovana rule3 funkcija sa generisanom
    funkcijom
rule3(in, (in: InputType, offs: Int) =>
  anonymous$1$(in, x, offs), x + 1, offs)
```

5.5 Evaluacija

Biblioteka zasnovana na makroima je evaluairana spram standardne Skaline biblioteke za kombinatore. Uz pomoć *Scalameter* alata za benčmarking, utvrđeno je da je parsiranje pomoću parser kombinatora ubrzano 19 puta. Primeri na kojima je vršeno testiranje su: CSV parser, JSON parser, parser DNK sekvenci... Svi fajlovi se prvo učitaju u $Array[Char]$ i prosleđuju parserima.

Skalina biblioteka za parser kombinatore koristi podrazumevani *CharSequence* da čita ulaz. Ova klasa uveliko korisiti *substring* metodu koja ima složenost

O(n) počev od verzije 7 Java virtualne mašine (u prethodnim verzijama je bila O(1), jer se string tretirao kao zaseban objekat što se u praksi nije pokazalo efikasnim zbog curenja memorije; u novijim verzijama string se "pod haubom"posmatra kao niska karaktera). Očigledno, korišćenje ove verzije *CharSequence*-a dovodi do velikog usporenja u parsiranju.

6 Zaključak

U vreme pisanja ovog rada *scala.reflect* predstavlja jedan od eksperimentalnih delova Skalinog kompajlera, koji je tu prisutan oko pet godina. Jedno od najvećih otvorenih pitanja u migraciji na Skalu 3.0 (u pitanju je verzija Skale, čiji izlazak na tržište se planira u 2020. godini) je: Šta uraditi sa makroima?

Prvi problem sa trenutnim def makroima je da su potpuno zavisni od tekućeg Skalinog kompajlera. Suštinski, oni su sintaksno ulepšanje zasnovano na internim svojstvima kompajlera. Ova osobina ih čini moćnim, ali i veoma teškim za upotrebu; zbog toga su i imali eksperimentalni status tokom svih godina rada na njima. Kako se ima u planu da Skala 3 koristi drugačiji kompajler, stari makro sistem, zasnovan na refleksiji neće moći da se na njega prebaci.

Potencijalni odgovor na uvodno pitanje, programeri koji rade na razvoju Skale vide u *Tasty* formatu. *Tasty* je format visokog nivoa u Skali 3 koji služi za razmenu koda koji je kompaktnom obliku. Zasniva se na tipiziranim AST-ovima, koja sadrže sve prisutne informacije o programu napisanom u Skali. Ova stabla predstavljaju sintaksnu strukturu programa i takođe sadrže kompletne informacije o tipovima i pozicijama. *Tasty* snimak fajla sa kodom (eng. *snapshot*) nastaje nakon provere tipova (tako da svi tipovi budu prisutni i da su svi impliciti razrešeni) ali pre bilo kakve transformacije (tako da se nijedna informacija ne izgubi ili promeni). Reprezentacija spomenutih stabala je višestruko optimizovana radi kompaktnosti, što znači da se mogu generisati u potpunosti *Tasty* stabla nakon svakog pokretanja kompajlera i da je dovoljno na njih se osloniti da bi se podržala odvojena kompilacija[15].

Ako se ova šema usvoji, ona će u velikoj meri odrediti izgled Skala 3 makroa. Najvažnije, oni će se izvršavati nakon faze provere tipova, jer su tek tada *Tasty* stabla spremna za upotrebu. Izvršavanje proširenja makroa nakon provere tipova ima mnoge prednosti:

1. bezbednije je i robustnije, jer je sve potpuno tipizirano
2. ne utiče na IDE, koji pokreće kompajler tek nakon završetke provere tipova
3. nudi veći potencijal za postepenu kompilaciju i paralelizaciju

Mana ovakvog pristupa je što ograničava tip makroa koji se mogu izraziti: makroi će biti *Blackbox*. To dalje povlači da ekspanzija makroa ne može uticati na tip proširenog izraza koji je određen od strane proveravča tipova. Sve dok je ovo ograničenje zadovoljeno, moguće je podržati klasične def makroe. Veruje se da se nedostatak *Whitebox* makroa može nadoknaditi izražajnijom formom izračunatih tipova.

Bibliografija

[1] The art of metaprogramming,
`https://www.ibm.com/developerworks/library/l-metaprog1/index.html`

[2] Quotations in OCaml,
`https://caml.inria.fr/pub/docs/tutorial-camlp4/tutorial004.html`

[3] Willard Van Orman Quine *Mathematical Logic* 1940

[4] Jim des Rivières, Brian C. Smith *The Implementation of Procedurally Reflective Languages* LFP '84 Proceedings of the 1984 ACM Symposium on LISP and functional programming, 1984

[5] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, Matthias Zenger *An Overview of the Scala Programming Language* École Polytechnique Fédérale de Lausanne (EPFL) 1015 Lausanne, Switzerland

[6] David Raymond Christiansen *Practical Reflection and Metaprogramming for Dependent Types* IT-Universitetet i København, 2016

[7] Implicits in Scala
`https://scala-lang.org/files/archive/spec/2.12/07-implicits.html`

[8] Eugene Burmako, Martin Odersky *Unification of Compile-Time and Runtime Metaprogramming in Scala* EPFL, 2017

[9] Denys Shabalin, Eugene Burmako, Martin Odersky *Quasiquotes for Scala, a Technical Report* EPFL, 2013

[10] Cake pattern in Scala
`https://medium.com/@itseranga/scala-cake-pattern-e0cd894dae4e`

[11] Scala meta
`https://scalameta.org/`

[12] Eric Béguet, Manohar Jonnalagedda *Accelerating Parser Combinators with Macros* EPFL, 2014

[13] Sylvain Marquis, Thomas R Dean, Scott Knight *Packet Decoding using Context Sensitive Parsing* 2005

[14] Wadler, Philip. *Monads for functional programming. Broy, Manfred (ed), Proc.* Marktoberdorf Summer school on program design calculi. Springer–Verlag, 1992

[15] Towards Scala 3
`https://www.scala-lang.org/blog/2018/04/19/scala-3.html`