Latvijas Lauksaimniecības universitāte

Latvia University of Agriculture

Informācijas tehnoloģiju fakultāte
Faculty of Information Technologies

Mg.sc.ing. **Mikus Vanags**

ABSTRAKTAS DATU APSTRĀDES TEHNOLOĢIJAS
ABSTRACT DATA PROCESSING TECHNOLOGIES

Promocijas darba KOPSAVILKUMS

inženierzinātņu doktora (Dr.sc. ing.) zinātniskā grāda iegūšanai
Informācijas tehnoloģijās

SUMMARY

of the Doctoral thesis for the scientific degree of Dr.sc. ing.

Jelgava 2017

Promocijas darba izstrāde: Latvijas Lauksaimniecības universitātē laika posmā no 2012. gada līdz 2017. gadam.

Doktora studiju programma: Informācijas tehnoloģijas.

Zinātniskā vadītāja: Dr.sc.comp. Rudīte Čevere.

Zinātniskā aprobācija promocijas darba fināla posmā:

- LLU ITF starpkatedru zinātniskajā seminārā 2017. g. 28. februārī.
- LLU informācijas tehnoloģijas nozares promocijas padomes sēdē 2017. g. 31. martā.

Oficiālie recenzenti

- Dr.CSc. Jaroslav Pokorný
- Dr.sc.comp. Kārlis Čerāns
- Dr.sc.ing. Andrejs Romānovs

Tulkojumu latviešu valodā veica: Mikus Vanags un Laura Aksika.

Promocijas darba aizstāvēšana notiks LLU Informācijas tehnoloģijas nozares promocijas padomes atklātajā sēdē 2017. gada 30. augustā plkst. 14:00, Jelgavā, Lielā ielā 2, Informācijas tehnoloģiju fakultātes 218. auditorijā.

Ar promocijas darbu un kopsavilkumu var iepazīties LLU Fundamentālajā bibliotēkā, Lielā ielā 2, Jelgavā LV-3001, un internetā (pieejams: http://llufb.llu.lv/promoc_darbi_en.html).

Atsauksmes sūtīt promocijas padomes sekretārei – Lielā ielā 2, Jelgavā, LV-3001; tālrunis: +371 63022584; e-pasts: tatjana.tabunova@llu.lv.

Padomes sekretāre: LLU lektore, Mg.paed. Tatjana Tabunova.

DOI: 10.22616/llu thesis/2017.010

SATURS

1 PUBLIKĀCIJAS UN DARBA PREZENTĒŠANA

Patenti:

- Vanags M., Justs J., Romanovskis D. (2015). ASV Patents: "IMPLICIT PARAMETERS AND IMPLICIT ARGUMENTS IN PROGRAMMING LANGUAGES". Publicēšanas datums: 21.05.2015. Publikācijas numurs: US 2015-0143330 A1. Patenta numurs: 9361071.
- Vanags M. (2016). ASV patents: "GRACE~ OPERATOR FOR CHANGING ORDER AND SCOPE OF IMPLICIT PARAMETERS". Publicēšanas datums: 14.07.2016. Publikācijas numurs: US 2016-0202955 A1. Patenta numurs: 9417850.

Patentu publikācijas bez patentiem:

- Vanags M., Licis A., Justs J. (2014). ASV Patenta pieteikums: "ABSTRACT, STRUCTURED DATA STORE QUERYING TECHNOLOGY". Publicēšanas datums: 20.03.2014. Publikācijas numurs: US 2014-0082014 A1. Patenta statuss: noraidīts.
- Vanags M., Licis A., Justs J. (2014). US Patent application: "STRONGLY TYPED METADATA ACCESS IN OBJECT ORIENTED PROGRAMMING LANGUAGES WITH REFLECTION SUPPORT". Publicēšanas datums: 06.03.2014. Publikācijas numurs: US 2014-0068557 A1. Patenta status: notiek vērtēšanas process.

Zinātniskās publikācijas:

- Vanags M., Licis A., Justs J. (2013). Strongly typed metadata access in object oriented programming languages with relection support. Baltic J. Modern Computing, Vol. 1 (2013), No. 1, p.77-100. Pieejams: http://www.bjmc.lu.lv/fileadmin/user_upload/lu_portal/projekti/bjmc/Contents/1_1-2_6_Vanags.pdf
- Vanags M., Licis A., Justs J. (2013). Meta-set calculus as mathematical basis for creating abstract, structured data store querying technology. 20th International Conference on Applications of Declarative Programming and Knowledge Management. Kiel, Germany, September 11-13, 2013. p.299-313. Pieejams: https://www.dcc.fc.up.pt/~ricroc/homepage/publications/2013-INAP.pdf (SCOPUS datubāzē).
- Vanags M., Cevere R. (2017). Type Safe Metadata Combining. Computer and Information Science; Vol. 10, No. 2; 2017. ISSN 1913-8989. Canadian Center of Science and Education. Pieejams https://doi.org/10.5539/cis.v10n2p97

Citas publikācijas:

- Vanags. M. Vanaga A. Mobilais matemātikas izpalīgs Il Calculus. Žurnāls "Skolas vārds" Nr. 27(69); 25.09.2014.

Dalība konferencēs:

- Vanags M., Licis A., Justs J. Meta-set calculus as mathematical basis for creating abstract, structured data store querying technology. 20th International Conference on Applications of Declarative Programming and Knowledge Management. Ķīlē, Vācijā, 2013.g. 11-13 septembrī.
- Vanags M. Software demonstration: "Kat - The Language of Calculations". ISSAC 2015 (International Symposium on Symbolic and Algebraic Computation). Bath, UK, 08.07.2015.
- Fundamental abstractions for data querying technologies. 6-th international scientific conference "Applied Information and Communication Technologies 2013", Jelgavā, 2013.g. 25-26 aprīlī.
- 8th International Scientific Conference "Students on Their Way to Science". Jelgavā, 24.05.2013.
- Implicit Lambda Calculus. Rēzeknes Augstskolas Inženieru fakultātes 18. Starptautiskajā student zinātniski praktiskajā konferencē "Cilvēks. Vide. Tehnoloģijas." Rēzeknē, 23.04.2014.
- Implicit parameters and implicit arguments in programming languages. Studentu zinātniski praktiskā conference "Jaunatnes loma un iespējas inženierzinātnes attīstībā". Daugavpilī, 08.05.2014.
- SIVA Koledžas 7. Zinātniski praktiskajā konferencē "Ekonomiskie un psiholoģiskie apsketi cilvēku ar invaliditāti izglītošanā un nodarbinātībā". Jūrmalā, 12.06.2014.
- Implicit parameters and implicit arguments in functional and logic programming. Rēzeknes Augstskolas Inženieru fakultātes 19. starptautiskā studentu zinātniski praktiskā konference "Cilvēks. Vide. Tehnoloģijas." Rēzeknē, 22.04.2015.
- Improvement possibilities of programming languages. Rēzeknes Augstskolas Inženieru fakultātes 19. starptautiskā studentu zinātniski praktiskā konference "Cilvēks. Vide. Tehnoloģijas." Rēzeknē, 22.04.2015.
- Type safe metadata access in programming languages. Rīgas Tehniskās Universitātes, Daugavpils filiāles konference. Daugavpilī, 23.04.2015.
- The perfect lambda syntax. 10th International Scientific Conference "Students on Their Way to Science". Jelgavā, 24.04.2015.
- Vanaga A., Vanags M. referāts "Mācību satura pilnveide, izmantojot funkcionālo programmēšanu". Dabaszinātņu un matemātikas skolotāju konferencē "Dabaszinātnes un matemātika skolā – efektīvi un radoši". Rīgā, 22.08.2014.

2 IEVADS

Programmatūras izstrādē un datorzinātnē abstrahēšana ir datorsistēmu sarežģītības slēpšanas tehnika. Lietotājs ar datorsistēmu darbojas augstā abstrakcijas līmenī, kur nav redzamas sistēmas sarežģītās detaļas. Programmētājs, savukārt, izmanto abstrakcijas, lai darbotos ar idealizētām saskarnēm, bez kurām darboties varētu būt pārāk sarežģīti (Abstraction, 2016).

Abstraktās datu apstrādes tehnoloģijas šajā darbā tiek interpretētas kā datu apstrāde, izmantojot vismaz vienu no trim šādām abstrakciju kategorijām:

1) Izteiksmīguma abstrahēšana:
   Dažkārt eksistējošās metodes un koncepti var nebūt pietiekami ērti, lai tos lietotu konkrētos gadījumos. Piemēram, mūsdienīgā matemātiskā pieraksta forma attīstījās neprognozējami (Stansifer R. D., 1994): simbolus "+" un "-" ieviesa Johans Vidmans (1489), "×" ieviesa Viljams Outreds (1631), "÷" ieviesa Johans Rahs (1659), utt.
   Izteiksmīguma abstrahēšanas piemērs ir jēdziena "funkcija" ieviešana, ko paveica Gotfrīds Leibnics savā 1673. gada vēstulē funkciju aprakstot kā vērtības piesaisti līknei konkrētā līknes slīpumā (Mastin L., 2010). Operācijas "+", "-", "×" un "÷" reprezentē katra vienu operāciju, bet funkcijas koncepts spēj reprezentēt gan vienkāršas operācijas, gan vienkāršu operāciju kombinācijas.
2) Nezināmas vērtības abstrahēšana:
   Dažkārt nākas darboties ar datiem, kuru vērtības vēl nav zināmas. Fransuā Vjets (1540-1603) risinot algebraiskas problēmas pirmais sistemātiski sāka apzīmēt nezināmus lielumus ar dažādiem simboliem (BashmakovaI G. et.al., 2000). Programmēšanas valodās līdzīgu abstrakciju dēvē par "mainīgo". Tas ļauj veikt simboliskos aprēķinus aizstājot nezināmos lielumus ar simbolu.
3) Bezgalīga koncepta abstrahēšana:
   Daži koncepti ir bezgalīgi, tādēļ, lai vispār būtu iespējams ar tiem darboties, ir nepieciešamas speciālas abstrakcijas. Teilora rinda ir summa bezgalīgas virknes elementiem, kurus veido funkcijas atvasinājumi konkrētos punktos (Kline M., 1990). Piemēram, funkcijas sin un cos var izteikt kā Teilora rindas (skat. **att. 2.1**). Par spīti funkciju sin un cos bezgalīgajai dabai, cilvēki izmanto šīs funkcijas. Tas ir iespējams pateicoties abstrakcijai - konkrētā gadījumā abstrakcija var būt aproksimācija. Šajā disertācijā netiek apskatītas citas bezgalīgu konceptu abstrakcijas, taču bezgalīgu konceptu abstrakcijas ir būtiska abstrakciju kategorija, kas uzskatāmi demonstrē, kādēļ dažkārt kaut kas nevar tikt paveikts bez atbilstošām abstrakcijām.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$$

att. 2.1. Sin and Cos funkcijas izteiktas kā bezgalīgas Teilora rindas

Pētāmās tēmas nozīmīgums: Vairums vispārējās nozīmes programmēšanas valodu ļauj izmantot lambda izteiksmes. Dažās valodās (JavaScript®, Swift™) lambda izteiksmes tiek sauktas par slēgumu (closure), taču tas nemaina šīs konstrukcijas semantisko jēgu: lambda ir matemātiskās funkcijas abstrakcija, kas programmēšanas valodās izskatās kā anonīmā funkcija. Ja lambda izteiksmju sintakses jomā tiktu izgudrotas jaunas izteiksmīguma abstrahēšanas abstrakcijas, kuras padarītu lambda sintaksi vēl kodolīgāku, tas pavērtu iespējas uzlabot eksistējošās programmēšanas valodas. Turklāt šādu tehnoloģiju prototips viedtālruņu pasaulē varētu padarīt ērtāku viedtālruņu lietošanu.

Tipu drošība ir nozīmīga tipu sistēmas īpašība (Pierce B.C., 2002). Modernajās programmēšanas valodās ir iestrādāts atbalsts konstrukcijām, notikumiem, atribūtiem (anotācijām) un citām struktūrām, bet eksistējošajās programmēšanas valodās, kurās iespējams izmantot refleksiju, vispār nav vai arī ir vājš atbalsts tipu drošai elementu metadatu apstrādei. Programmētāju produktivitāti varētu paaugstināt, ja tiktu izgudrota jauna izteiksmīguma abstrahēšanas metode kā programmēšanas valodās uzlabot tipu drošību metadatu apstrādi.

Programmētāji ikdienā izmanto dažādas abstrakcijas, lai darbotos ar dažādām datu izgūšanas tehnoloģijām un dažādām datu izgūšanas valodām. Pašlaik neeksistē viena abstrakta datu izgūšanas tehnoloģija, kura būtu piemērota jebkādu strukturētu datu avotu apstrādei. Šāda abstrakta datu vaicājumu tehnoloģiju varētu izmantot nezināmas vērtības abstrahēšanu, lai panāktu neatkarīgumu no konkrēta datu avota arhitektūras. Viena abstrakta datu izgūšanas tehnoloģija nozīmētu, ka tiek samazināts programmētāju apmācības laiks un izmaksas, lai mācētu lietot vienu, universālu datu izgūšanas tehnoloģiju kā arī programmētāji datu izgūšanas jomā varētu darboties produktīvāk.

Jaunu abstrakciju meklēšana un inovāciju radīšana programmēšanas valodu jomā ir svarīga, jo tas ietekmē mūsu domāšanu par problēmām: “Valoda un tās pieraksts, ko mēs lietojam, lai izteiktu vai pierakstītu savas domas, ir galvenais faktors, kas ierobežo par ko mēs spējam domāt un ko vispār spējam izteikt!” - Edsgers W. Deikstra.

Pētījuma mērķis: atrast jaunas abstrakcijas datu apstrādes jomā un izstrādāt lietotājam draudzīgākus datu apstrādes tehnoloģiju prototipus.

Pētījuma tēzes: eksistē jaunas abstrakcijas datu apstrādes jomā:

1) simbolisko aprēķinu jomā ar tām iespējams padarīt lambda izteiksmes īsākas;
2) metadatu apstrādes jomā ar tām iespējams uzlabot tipu drošību.

Darba uzdevumi:

1) Uzlabot simbolisko aprēķinu tehnoloģijas;
2) Izpētīt uzlaboto simbolisko aprēķinu pielietošanas iespējas;
3) Izveidot prototipu aprēķinu veikšanai ar uzlabotajām simbilosko aprēķinu tehnoloģijām;
4) Uzlabot tipu drošas metadatu apstrādes tehnoloģijas;
5) Izpētīt uzlaboto metadatu apstrādes tehnoloģiju pielietojuma iespējas;
6) Izveidot prototipu metadatu apstrādei ar uzlabotajām metadatu apstrādes tehnoloģijām.

Izmantotās pētniecības metodes:

1) Literatūras, interneta avotu un patentu analīze (kā daļa no ASV patentu pieteikumu 14594113, 14081460, 13781804, 13773662 ekspertīzes procesa. Informācija publiski pieejama USPTO public pair sistēmā (USPTO, 2017)).
2) Novērošana, salīdzināšana un dedukcija – salīdzinot dažādus faktorus (koda lasāmību, rediģējamību, izteiksmīgumu un kodolīgumu) un analizējot dažādas metodes lambda sintakses izveidē un tipu drošā metadatu apstrādē.
3) Ideju ģenerēšana (intelektuālas diskusijas).
4) Decentralizētā dedukcijas dziņa prototipa izstrāde un integrācija "sia Sparlats" ogu uzskaites programmatūrā (2013-2014).
5) Abstraktas aprēķinu programmēšanas valodas Kat-lang dizaina izveide.
6) Pielāgojama, lietotājam draudzīga kalkulatora prototipa izstrāde Windows Phone 8.1 platformai un jaunākām versijām. (IlCalculus, 2015). Iekļauj simulāciju – izstrādātā prototipa testēšanu dažādu viedtālruņu modeļu emulatoros ar dažādiem ekrānu izmēriem.

Darba zinātniskie jauninājumi:

1) Atklāti netiešie parametri, netiešie argumenti un netiešo parametru secības korekcijas operators Grace~ (visas minētās inovācijas veiktas sintaksē, nemainot parametru un argumentu semantiku).
2) Izmantojot netiešos parametrus un Grace~ operatoru, definēta lambda izteiksmju perfektā sintakse.
3) Izveidota abstrakta programmēšanas valoda Kat (KatLang) aprēķinu veikšanai.
4) Atklāts tipu drošs metadatu apstrādes operators "memberof", kontekstatkarīgais metadatu apstrādes operators "member" un parametru

modifikators “meta”, kas ļauj uzlabot tipu drošību metadatu apstrādes operācijās.

5) Atklāta meta-kopas abstrakcija, kas abstrahē objektu kopu un ir meta-kopu rēķinu pamats. Darbā definētie meta-kopu rēķini ir otrās kārtas predikātu loģikas paveids, kas ļauj veikt dedukciju decentralizēti – ar datu avotu nesaistītā vidē.

Lai uzskatāmāk parādītu, kuras koda daļas attiecas uz darba gaitā radītajām inovācijām, darba gaitā radītās inovācijas koda fragmentos tiek attēlotas sarkanā krāsā:

```
inovācija;
```

Sarkanā krāsa izvēlēta, jo eksistējošo programmēšanas valodu kompilatori nezināmās konstrukcijas attēlo sarkanā krāsā. Darba inovācijas nav izceltas sarkanā krāsā nodaļās 0, 7, un 8, jo šajās nodaļās aplūkoti jaunu programmēšanas valodu koncepti un konceptuāli jauni veidi datu izgūšanā.

3 NETIEŠIE PARAMETRI UN NETIEŠIE ARGUMENTI

Termins “parametrs” jeb “formālais parametrs” tiek attiecināts uz mainīgo, kas ir daļa no funkcijas definīcijas. Savukārt, termins “arguments” jeb “aktuālais parametrs” tiek attiecināts uz ievades datiem, kas tiek nodoti funkcijai tās izpildes laikā (Stansifer R. D. (1994). Dažās programmēšanas valodās termina “parametrs” vietā tiek lietots termins “arguments” un otrādi. Tā kā nav vienkārši panākt, lai vienmēr vienas un tās pašas lietas tiktu sauktas vienos un tajos pašos vārdos, tad šajā promocijas darbā dažādu abstrakciju nosaukumi ir lietoti to oriģinālajā formā. Līdz ar to, redzot tādus nosaukumus kā “netiešais parametrs”, “parocīgais argumenta nosaukums” u.c. ieteicams pievērst uzmanību koda piemēriem, kas ir vienīgais patiesības avots, lai noskaidrotu, par ko iet runa: par parametru (funkcijas definīcijas sastāvdaļu) vai argumentu (funkcijas izsaukuma sastāvdaļu).

3.1 Netiešie parametri

Netiešais parametrs ir tāds parametrs, kas ir deklarēts metodes ķermenī (skat. **att 3.1**), nevis metodes galvgalī, kā tas parasti notiek ar parametriem.

Netiešo parametru deklarācijas sintakse ir šāda:

```
ParametraTips parametraNosaukums
```

Netiešo parametru deklarācijas sintakse ir līdzīga lokālo mainīgo deklarācijas sintaksei, kur lokālā-mainīgā-deklarators tiešā vai netiešā veidā norāda mainīgā tipu, bet identifikators - mainīgā nosaukumu:

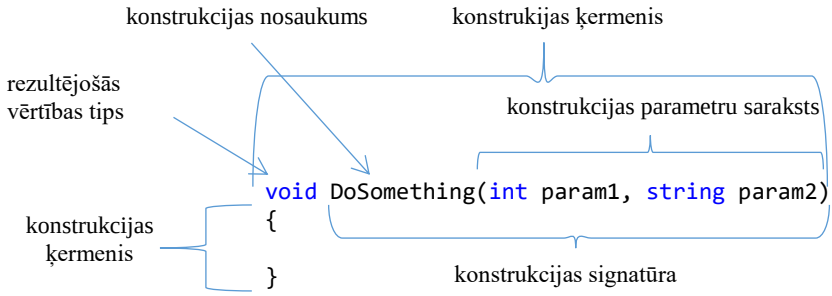
```
lokālā-mainīgā-deklarators identifikators;
```

Lokālie mainīgie tiek deklarēti kā teikumi (kods, kas kaut ko dara), bet netieši parametri tiek deklarēti kā izteiksmes (kods, kura izpildes rezultātā tiek iegūta vērtība). Netiešie parametri kā izteiksmes var tikt lietoti:

- teikumos;
- citās izteiksmēs (kā daļa no lielākas izteiksmes).

Lokālo mainīgo iespējams deklarēt kopā ar tā inicializācijas izteiksmi. Piemērs, kur "inicializācija" ir mainīgā inicializācijas izteiksme:

```
lokālā-mainīgā-deklarators identifikators = inicializācija;
```



att 3.1. Funkcijas deklarācijas piemērs valodā C#

Piemērs, kur konstrukcijas netiešais parametrs "param1" ir deklarēts kā lokālā mainīgā inicializācijas izteiksme:

```
//parametrs param1 deklarēts netieši
void DoSomething2() {
    int x = int param1;
    Console.WriteLine(x);
}
```

Izmantojot tiešā veidā deklarētus parametrus, "DoSomething2" iespējams pārrakstīt šādi:

```
void DoSomething2(int param1) {
    int x = param1;
    Console.WriteLine(x);
}
```

Tā kā netiešā parametra tipu iespējams noteikt no netiešā parametra deklarācijas konstrukcijas ķermenī, tad netiešo parametru ideja nemaina konstrukcijas izsaukuma sintaksi. Piemērā attēlots konstrukcijas "DoSomething2" izsaukums, kas nemainās neatkarīgi no tā vai parametrs tika deklarēts tiešā vai netiešā veidā:

```
//konstrukcijas izsaukumā argumenti jānorāda obligāti
DoSomething2(6);
```

Parastie parametri tiek iekļauti konstrukcijas parametru sarakstā pirmie, pēc tam seko netiešie parametri. Ja konstrukcija satur vairākus netiešos parametrus, to secība parametru sarakstā atbilst parametru izkārtojumam konstrukcijas ķermenī.

Nākamajā piemērā parādīta netiešo parametru deklarēšana konstrukcijas izpildes izteiksmes teikumā (ja sintakstiskā konstrukcija nerezultējas vērtībā vai arī tā rezultējas vērtībā, bet vērtība netiek izmantota, tad konstrukcija tiek interpretēta kā teikums, pretējā gadījumā – izteiksme):

```
void DoSomething5() {
    int x = int param1;
}
void DoSomething6() {
    //netiešā parametra deklarēšana konstrukcijas izpildē
    DoSomething5(int param1);
}
```

Izmantojot tiešā veidā deklarētus parametrus, "DoSomething5" un "DoSomething6" iespējams pārrakstīt šādi:

```
void DoSomething5(int param1) {
    var x = param1;
}
void DoSomething6(int param1) {
    DoSomething5(param1);
}
```

Līdzīgā veidā netiešos parametrus iespējams deklarēt kā daļu no teikumiem: "if", "switch", "while", "for", u.c. Netiešos parametrus iespējams deklarēt arī kā daļu no kompleksām izteiksmēm (binārās izteiksmes, piešķires izteiksmes, unārās izteiksmes, u.c. izteiksmēm, kuru deklarācijā jādefinē vismaz viena jauna izteiksme).

Netiešos parametrus var izmantot visās konstrukcijās, taču jārēķinās, ka netiešie parametri maina konstrukcijas signatūru. Tas nozīmē, ka netiešo parametru lietošana, piem., programmas ieejas punktā (konstrukcijā "Main") var radīt kompilācijas kļūdas.

Dažkārt parametrs konstrukcijas ķermenī tiek lietots tikai vienreiz un netiek izmantots atkārtoti. Šādos gadījumos ērti lietot netiešo parametru anonīmo formu, kuras sintakse ir šāda:

```
ParametraTips
```

ParametraTips nosaka netiešā anonīmā parametra tipu, bet parametra nosaukums netiek norādīts, jo to kompilators ģenerēs automātiski.

```
//anonīmā netiešā parametra izmatnošana
void DoSomething7() {
    int x = int param1;
    int y = param1; //y == x
    int z = int; //anonīmais netiešais parametrs
}
```

Izmantojot tiešā veidā deklarētus parametrus, “DoSomething7” iespējams pārrakstīt šādi:

```
void DoSomething7(int param1, int autoGeneratedParamName1) {
    int x = param1;
    int y = param1; //y == x
    int z = autoGeneratedParamName1;
}
```

Bieži netiešo parametru tipu iespējams noteikt no to izmantošanas konteksta Šādos gadījumos netiešo parametru sintaksi iespējams uzlabot atsakoties no tipa deklarēšanas netiešo parametru sintaksē. Tad saīsinātā sintakse tiks saukta par netiešo parametru kanonisko formu, kura izskatās šādi:

```
parametraNosaukums
```

Netiešā parametra kanoniskās formas izmantošanas piemērs piešķires operatora teikumā ir šāds:

```
void DoSomething11() {
    //netiešā parametra kanoniskā forma
    int x = param1;
}
```

Izmantojot tiešā veidā deklarētus parametrus, “DoSomething11” iespējams pārrakstīt šādi:

```
void DoSomething11(int param1) {
    var x = param1;
}
```

Netiešā parametra kanoniskās formas izmantošana argumenta izteiksmē konstrukcijas izpildes teikumā:

```
void DoSomethingWithX(int x) { }
void DoSomething13() {
    //netiešā parametra kanoniskā forma
    DoSomethingWithX(param1);
}
```

Izmantojot tiešā veidā deklarētus parametrus, "DoSomething13" iespējams pārrakstīt šādi:

```
 void DoSomething13(int param1) {
    DoSomethingWithX(param1);
}
```

Kompilējot kodu, var nākties saskarties ar problēmu: "Kā interpretēt nezināmos identifikatorus un ko ar tiem darīt?" Vienkāršākais risinājums: pārtraukt programmas kompilāciju ar kļūdas paziņojumu. Ja tiek izmantoti netiešie parametri, tad nav jādomā, kā interpretēt nezināmos identifikatorus konstrukcijas ķermenī, jo tie visi tiks interpretēti kā konstrukcijas netiešie parametri.

3.2 Netiešo parametru secības maiņa, lietojot Grace~ operatoru

Netiešo parametru secība konstrukcijas parametru sarakstā atbilst netiešo parametru deklarācijas secībai konstrukcijas ķermenī. Taču dažkārt būtu ērti, ja parametri konstrukcijas parametru sarakstā tiktu izkārtoti savādāk nekā tie ir tikuši deklarēti. Nākamais piemērs ilustrē minēto situāciju (tiek lietoti tiešā veidā deklarēti parametri):

```
void PrintName(string firstName, string lastName) {
    Console.WriteLine(lastName);
    Console.WriteLine(firstName);
}
```

Situāciju iespējams labot ieviešot Grace~ operatoru, kas ļauj mainīt parametru secību (Vanags M., 2015). Grace~ operatora simbols ir tildes simbols, bet ieteicamā rakstība dabiskajā valodā: "Grace~" nevis "~".

Grace operatora sintakses formas ir šādas:

1) Prefiksā forma (Grace~ operators tiek lietots pirms netiešā parametra):

```
~parametrs
```

Grace~ operatora prefiksā forma pārvieto parametru par vienu pozīciju tuvāk konstrukcijas parametru saraksta sākumam:

```
function PrintName() {
    Write(lastName);
    Write(~firstName);
}
```

Rezultātā parametrs "firstName" konstrukcijas parametru sarakstā tiek apmainīts vietām ar parametru "lastName", respektīvi – mainās parametru atrašanās vieta konstrukcijas parametru sarakstā.

2) Postfiksā forma Grace~ operators tiek lietots tieši aiz netiešā parametra):

```
parameters~
```

Grace~ operatora postfiksā forma pārvieto parametru par vienu pozīciju tuvāk konstrukcijas parametru saraksta beigām:

```
function PrintName() {
    Write(lastName~);
    Write(firstName);
}
```

Rezultātā parametrs "lastName" konstrukcijas parametru sarakstā tiek apmainīts vietām ar parametru "firstName".

Grace~ operators neietekmē algoritmu, kas tiek definēts konstrukcijas ķermenī. Grace~ operators maina tikai netiešo parametru sacību konstrukcijas parametru sarakstā.

Viens Grace~ operatora pielietojums liek kompilatoram konstrukcijas parametru sarakstā veikt vienu parametru samainīšanas operāciju. Grace~ operatora prefiksās formas gadījumā samainīšanas operācija tiek veikta ar iepriekšējo parametru, bet postfiksās formas gadījumā – ar nākošo parametru. Tāpat tas nozīmē, ka Grace~ operatora divkārtīga lietošana nozīmē divas parametru samainīšanas operācijas konstrukcijas parametru sarakstā.

Pieņemsim, ka eksistē konstrukcija "GetValue", kas deklarēta izmantojot tiešos parametrus:

```
function GetValue(a, b, c) {
    return c - b + a;
}
```

Lietojot netiešos parametrus un Grace~ operatoru, konstrukciju "GetValue" iespējams deklarēt šādi:

```
function GetValue() {
    return c~~ - b + ~a;
}
```

Eksistē dažādi veidi kā sasniegt vēlamo rezultātu. Ekvivalents piemērs iepriekšējam:

```
function GetValue() {
    return c~ - b + ~~a;
}
```

Kā arī viens piemērs, ar pārmērīgu Grace~ operatora lietošanu:

```
function GetValue() {
    return c~~~ - ~b~ + ~a;
}
```

Iepriekšējo piemēru iespējams sadalīt apakšizteiksmēs šādi:

1) c~~~ pārvieto parametru ‘c’ divas pozīcijas tuvāk konstrukcijas parametru saraksta beigām līdz tiek sasniegtas saraksta beigas, tādēļ trešais Grace~ operators tiek ignorēts. Pēc apakšizteiksmes apstrādes konstrukcijas parametru saraksts izskatās šādi: [b, a, c];
2) ~b~ nemaina konstrukcijas parametru sarakstu.
3) ~a pārvieto parametru vienu pozīciju tuvāk konstrukcijas parametru saraksta sākumam samainot vietām “a” ar “b”. Rezultātā konstrukcijas parametru saraksts izskatās šādi: [a, b, c].

Vairākkārtīgs Grace~ operatora lietošanas efekts summējas. Operatora lietošana tiek ignorēta tikai, ja vairs nav iespējams veikt parametru samainīšanas operācijas konstrukcijas parametru sarakstā.

Aplūkosim šādu piemēru:

```
function SumOfParams() {
    return a~~ + b~;
}
```

Izteiksmes “a~~” pirmais Grace~ operators pārvieto parametru “a” uz konstrukcijas parametru saraksta beigām, bet otrais Grace~ operators šoreiz netiek ignorēts, tas tiks neitralizēts ar izteiksmes ”b~” Grace~ operatoru. Tādējādi parametrs “b” netiek pārvietots uz konstrukcijas parametru saraksta beigām.

Izmantojot tiešā veidā deklarētus parametrus, konstrukciju “SumOfParams” iespējams pārrakstīt šādi:

```
function SumOfParams(b, a) {
    return a + b;
}
```

Lai padarītu ērtāku daudzkārtīgu Grace~ operatora lietošanu, eksistē šādas Grace~ operatora papildus sintaktiskās formas:

1) Prefiksā forma:

```
N~parameters
```

2) Postfiksā forma:

```
parameters~N
```

kur N ir konstante, kas nosaka cik reizes Grace~ operators tiek pielietots.

Tā kā Grace~ operators nemaina konstrukcijas algoritmu konstrukcijas ķermenī, tad Grace~ operatoram jābūt lielākai prioritātei nekā pārējiem konstrukcijas ķermenī lietotajiem operatoriem. Tādēļ, lai neietekmētu konstrukcijas algoritmu, konstrukcija:

```
function CalculateSomething() {
    return b + c + 2~a;
}
```

interpretējama šādi:

```
function CalculateSomething(a, b, c) {
    return b + c + a;
}
```

nevis šādā nekompilējamā formā:

```
function CalculateSomething(???) {
    /* Grace~ operatora lietošana nosaka, ka '(c+2)' jābūt
konstantei, pretējā gadījumā izteiksme nav valīda */
    return b + (c + 2)~a;
}
```

3.3 Netiešie argumenti

Konstrukciju izpildēs iespējams nenorādīt neobligātos argumentus, bet nav iespējams izvairīties no obligāto argumentu norādīšanas. Turklāt, neobligātos argumentus iespējams izmantot tikai tad, ja tiem atbilstošo parametru deklarācija satur argumenta noklusēto vērtību. Tas nozīmē, ka, lai

izmantotu neobligātos argumentus, to lietojums ir jāparedz deklarējot konstrukciju ar tās parametriem. Atsevišķos gadījumos var būt nepieciešama piekļuve konstrukcijas deklarācijai ar tiesībām to mainīt, taču šādas tiesības ne vienmēr iespējams iegūt. Risinājums, kā nenorādīt obligātos argumentus konstrukciju izpildē ir netiešo argumentu inovācija (Vanags M. et.al., 2013).

Ja konstrukcijas izpildē obligātie argumenti nav norādīti, tad izlaistie argumenti tiek automātiski pievienoti konstrukcijas, kurā notiek izsaukums, parametru sarakstam, tādējādi izmainot konstrukcijas signatūru:

```
void DoSomething17(int param1) {
    int x = param1;
    int y = int param2;
}
void DoSomething18() {
    //param1, param2 tiek pievienoti DoSomething18 parametriem
    DoSomething17();
}
void DoSomething19(int paramA, int paramB) {
    /*šīs izpildes sintakse var tikt uzlabota, ja nomainam
parametru nosaukumus paramA un paramB uz param1 un param2. */
    DoSomething18(paramA, paramB);
}
```

Izmantojot tiešā veidā deklarētus parametrus un argumentus, iepriekšējais piemērs var tikt pārveidots šādi:

```
void DoSomething17(int param1, int param2) {
    var x = param1;
    var y = param2;
}
void DoSomething18(int param1, int param2) {
    DoSomething17(param1, param2);
}
void DoSomething19(int paramA, int paramB) {
    DoSomething18(paramA, paramB);
}
```

Konstrukcijas “DoSomething19” parametru nosaukumi atšķiras no konstrukciju “DoSomething18” un “DoSomething17” parametru nosaukumiem. “DoSomething18” izpildes sintakse konstrukcijas DoSomething19 ķermenī var tikt uzlabota, ja nomainām konstrukcijas “DoSomething19” parametru nosaukumus “paramA” un “paramB” ar “param1” un “param2”. Uzlabotais piemērs izskatās šādi:

```
void DoSomething20(int param1) {
    int x = param1;
    int y = int param2;
}
void DoSomething21() {
// DoSomething20 izpildes argumenti noteikti automātiski un
// pievienoti DoSomething21 parametru sarakstam
    DoSomething20();
}
void DoSomething22(int param1, int param2) {
/*ja konstrukcijas parametru nosaukumi sakrīt ar tiem kādi
izmantoti iepriekšējā piemērā DoSomething21 izpildē, tad
DoSomething21 argumentus var nenorādīt, tie var tikt noteikti
automātiski*/
    DoSomething21();
}
```

Konstrukcija "DoSomething22" ir ekvivalenta šādai konstrukcijai, kurā izmantoti netiešie argumenti "DoSomething21" izpildē un netiešie parametri "DoSomething22" deklarācijā:

```
/*konstrukcija satur netiešos parametrus: int param1, int param2
void DoSomething22() {
    /*DoSomething21 argumenti tiek pievienoti DoSomething22
parametru sarakstam*/
    DoSomething21();
}
```

Lietojot netiešos argumentus, parametru nosaukumi kļūst par lietojumprogrammu saskarnes (API) sastāvdaļu un atsevišķos gadījumos tas var novest pie nevēlama rezultāta un kļūdām. Aplūkosim šādu piemēru:

```
void DoSomething23() {
    int x = int param1;
    int y = int param2;
}
void DoSomething24() {
    int x = int param1;
    float y = int param2;
}
void DoSomething25() {
    DoSomething23();
    DoSomething24();
}
```

Konstrukcija “DoSomething25” saturēs 3 netiešos parametrus, no kuriem diviem būs vienāds nosaukums - “param2”, bet dažādi tipi. Šādas situācijas nav pieļaujamas.

Netiešo argumentu nosaukumu konfliktu iespējams atrisināt lietojot dažādus netiešos parametrus, piem., šādi:

```
void DoSomething25() {
    DoSomething23(int param1, int param2);
    //Šeit parametrs param1 tiek izmantots atkārtoti.
    DoSomething24(param1, float param3);
}
```

Ekvivalents konstrukcijas “DoSomething25” piemērs, neizmantojot netiešos parametrus un netiešos argumentus, izskatās šādi:

```
void DoSomething25(int param1, int param2, float param3) {
    DoSomething23(param1, param2);
    DoSomething24(param1, param3);
}
```

4 PERFEKTĀ LAMBDA SINTAKSE

Pirmo reizi lambda tika ieviesta „Lambdas aprēķinos” (Rojas R., 1998), to izgudroja matemātiķis Alonso Čērčs 1930-tajos gados, un tā ir tikusi izmantota Lisp programmēšanas valodā kopš 1958.gada (Veitch J., 1998). Lambda ir matemātiskās funkcijas abstrakcija, kas programmēšanas valodās izteikta anonīmo funkciju veidā (Душкин Р.В., 2006).

```
function (x) { return x * x }; //anonīmās funkcijas piemērs
```

Lambdu bieži izmanto kā izteiksmi (kaut kas, kas rezultējas vērtībā), tādējādi to sauc par „lambdas izteiksmi”. Izmantojot Lambda izteiksmes parasto funkciju vietā, programmētājiem nav jāraizējas:

- kā atdalīt funkcijas deklarāciju no tās izpildes;
- kā nosaukt kaut ko, kas tiek izmantots tikai vienu reizi.

Lambda izteiksmēm ir sintaktiska priekšrocība pār tradicionālo anonīmo funkciju deklarēšanu: lambda izteiksmes ļauj atbrīvojoties no tādiem artefaktiem kā *function* un *return* atslēgvārdiem, kas lambda izteiksmes padara līdzīgas izpildāmiem koda blokiem (Atencio L., 2015).

```
x => x * x
```

Šī sintakse kodu padara kodolīgāku, ja tā tiek lietota kopā ar augstākas kārtas funkcijām, kā, piem., *map*, *reduce* un *filter*:

```
range(1, 5).map(x => x * x); //-> [1, 4, 9, 16, 25]
```

Lambdas iedrošina praktizēt pirmo no slavenajiem programmatūras izstrādes principiem: "Vienīgā atbildība" (Martin R.C., 2003). Attiecināts uz funkcijām, šis "Vienīgās atbildības" princips nozīmē, ka funkcijām jābūt tikai vienai atbildībai. Lambda izteiksmes var saturēt vairākus teikumus:

```
   (x, y) => {
    // do something with x and y
    // statement 1
    // statement 2
    return result;
}
```

Bet lielākoties to pielietojums ir atvasināts no ierindošanas (inlined) kā funkcijas argumentam garākās izteiksmēs vai kompozīcijā. Tas nozīmē, ka ir iespējams uzrakstīt vairākas mazas lambda funkcijas, kas dara precīzi vienu lietu, un sakombinēt tās, veidojot kopīgas programmas:

```
range(1, 5)
    .filter(x => (x % 2) === 0)
    .map(x => x * x);    //-> [4, 16]
```

Lambdas izteiksmes iedrošina radīt programmas, kas ir deklaratīvas, modulāras un atkārtoti lietojamas pat ļoti detalizētā līmenī. Bet izrādās, ka eksistējošā lambda izteiksmju sintakse nav perfekta un tā var tikt uzlabota.

Programmētāji visbiežāk lasa programmēšanas valodas kodu. Tādējādi koda lasāmība ir svarīgs perfektās lambda sintakses raksturlielums. Programmētāji kodu arī raksta un labo. Rakstīt kodu ir vieglāk, ja jāuzraksta mazāk simbolu. Tas pats attiecas arī uz koda labošanu. Arī koda lasāmība ir atkarīga no simbolu skaita, ko tas satur. Ir vēl citi faktori, kas var ietekmēt koda lasāmību, piemēram, ekrāna izmērs, teksta fonts, testa krāsa, fona krāsa, atstarpju lietojums utt. Katram no mums var būt savs iecienītākais ekrāna izmērs, teksta izmērs un krāsa, bet simbolu skaits tiek interpretēts nepārprotami. Līdz ar to, lambda izteiksmju sintakse var tikt saukta par perfektu, ja tā tiek izteikta viskodolīgākajā formā, nemainot iecerētajai izteiksmei semantiku jeb jēgu. Daži varētu iebilst, ka vārdu skaitīšana ir labāka par simbolu skaitīšanu, jo identifikatori var tikt izpausti informatīvāk. Piemēram, identifikators 'personX' ir aprakstošāks nekā 'x'. Tādējādi šajā pētījumā tiek piedāvātas divas dažādas perfektās lambda sintakses formas:

- simbolu perfekta lambda sintakse – lambda sintakse, kas satur iespējami mazāku skaitu simbolu;
- vārdu perfekta lambda sintakse – lambda sintakse, kas satur iespējami mazāku vārdu skaitu.

Kamēr vien vārdi un simboli ir noderīgi un neatkārto lambda izteiksmju definīcijā jau eksistējošu informāciju, tie tiek uzskatīti par nozīmīgu daļu no perfektās lambda sintakses. Ja lambdas izteiksmes ir vienkāršas un to parametru nozīme var tikt saprasta no lietojuma konteksta (teksta saprotamība ir laba un neprasa papildu informāciju), tad perfektā lambdas sintakse var būt simbolu perfekta. Ja lambdas sintakse ir simbolu perfekta, tad acīmredzami tā ir arī vārdu perfekta.

Pēdējo pāris gadu laikā lambdas izteiksmes ir kļuvušas par populāru valodu konstrukciju un tā ir ieviesta teju katrā komerciālajā programmēšanas valodā. Dažās valodās (JavaScript, Swift) lambdas izteiksmes sauc par slēgumiem (closures), Kotlin lambdas izteiksmes dēvē par "funkciju literāļiem". Pastāv daudz nosaukumu vienām un tām pašām semantiskajām konstrukcijām, tāpēc vienīgais veids, kā objektīvi spriest par lambdām dažādās programmēšanas valodās, ir salīdzināt koda fragmentus no dažādām lambda sintaksēm.

Lambda izteiksmju uzlabojumi, izmantojot neapšaubāmu parametru pilnā formā un atslēgvārdu 'func', lai norādītu lambda izteiksmes sākumu, ir nodemonstrēti šādā piemērā:

```
func (Person x).LastName.CompareTo((Person y).LastName)
```

Lietojot tipu inferenci, kompilators var noteikt parametru tipus no metodes 'CompareTo' signatūras. Tādējādi, iepriekšējais piemērs var tikt uzlabots, atmetot parametra tipa deklarāciju un pielietojot netiešos parametrus kanoniskajā formā, kā parādīts nākošajā piemērā:

```
func x.LastName.CompareTo(y.LastName)
```

Tā kā parametri 'x' un 'y' lambdas ķermenī netiek lietoti atkārtoti, tie var tikt deklarēti anonīma netiešā parametra formā šādi:

```
funcPerson.LastName.CompareTo(Person.LastName)
```

Ja vēlme ir sarindot cilvēku sarakstu pretējā secībā, tad lambdas parametru kārtība būtu jāmaina, un to var paveikt, lietojot *Grace*~ operatoru prefiksajā formā, kā redzams šādā piemērā:

```
people.Sort(funcPerson~.LastName.CompareTo(Person.LastName));
```

vai lietojot *Grace*~ operatoru postfiksajā formā:

```
people.Sort(funcPerson.LastName.CompareTo(~Person.LastName));
```

Ja lambdas ķermenis sastāv no vairāk kā viena izteiksmes-teikuma, tad lambdas deklarācijā var tikt izmantots koda bloks, kā tas ir redzams šādā piemērā:

```
people.Sort(func {
returnPerson.LastName.CompareTo(Person.LastName);
    });
```

Salīdzinot konstrukciju norādes ('method references') piemēru:

```
people.sort(comparing(Person::getLastName));
```

ar piemēru, demonstrējot *netiešo parameteru* lietojumu:

```
people.sort(comparing(func Person.getLastName()));
```

izskatās, ka konstrukciju norāžu sintakse fokusējas uz sintakses kodolīgumu, bet konstrukciju norādes pielietojamas tikai gadījumos, kad lambda izteiksmes ķermenis sastāv no vienas konstrukcijas izpildes, līdz ar to metožu norāžu pielietojums ir ļoti ierobežots. Netiešajiem parametriem nav tāda ierobežojuma.

Swift programmēšanas valodai ir ievērojami valodas dizaina ierobežojumi, lietojot “parocīgos argumentu nosaukumus” (shorthand argument names) iestarpinātās lambdas izteiksmēs. Lai pārvarētu šo ierobežojumu, programmētājiem ir jāpielieto dažādi paņēmieni. Nākamais piemērs demonstrē iestarpinātu funkciju ar "parocīgo argumenta nosaukumu" lietojumu iekšējā (iestarpinātajā) anonīmā funkcijā:

```
var outerFunc2: (Int) -> Int =
{
    var outer = $0
    var innerFunc: (Int) -> Int = { outer + $0 }
    return innerFunc(5)
}
```

Netiešie parametri pretēji "parocīgajiem argumentu nosaukumiem" var tikt brīvi lietoti abos konstrukciju sasniedzamības līmeņos: gan ārējā, gan iestarpinātajā anonīmajā funkcijā. Lietojot netiešos parametrus kopā ar parametru secības un sasniedzamības līmeņa korekcijas operatoru Grace~, kods var tikt padarīts vēl kodolīgāks un lasāmāks, kā tas tiek demonstrēts šādā piemērā:

```
var outerFunc2: (Int) -> Int =
{
    var innerFunc: (Int) -> Int = { ~outer + inner }
    return innerFunc(5)
}
```

Netiešie parametri un Grace~ operators ir nepieciešami elementi, lai iegūtu perfekto lambdas sintaksi. Perfekta lambda sintakse nevar saturēt tiešu funkcijas parametru saraksta deklarāciju, tāpēc perfektajā lambda sintaksē tiks

lietoti netiešie parametri. Savukārt, Grace~ operators patērē mazāku skaitu simbolu un vārdu, lai mainītu parametru secību nekā lietojot tiešu lambdas parametru saraksta deklarāciju.

Netiešajiem parametriem ir viens dizaina ierobežojums – tie nespēj definēt konstrukcijas, kas satur parametru, bet neizmanto to konstrukcijas ķermenī. Konstrukcija, kas attēlota nākamajā koda fragmentā, nevar tikt definēta, lietojot vienīgi netiešos parametrus bez papildus abstrakcijām:

```
function PrintPartOfName(string lastName, string firstName) {
    Write(firstName);
}
```

Ja lambda satur parametru, kas nav lietots lambdas ķermenī, tad šāda lambda sintakse satur liekvārdību, nav perfekta un var tikt uzlabota.

Daži neatbildēti jautājumi ir šādi: „Vai perfektā lambda satur lambdas simbolu? Varbūt lambdas sintakse var sastāvēt tikai no tās ķermeņa deklarācijas (kuram arī jābūt perfektam)?" Swift "parocīgo argumentu nosaukumu" lietojums demonstrē, ka lambdas simbols var tikt noteikts no lambdas lietojuma konteksta. Tādējādi lambda izteiksme, kas satur lambdas simbolu, nav perfekta saskaņā ar perfektās lambda sintakses definīciju. Vispārējās nozīmes programmēšanas valodās varētu būt nepieciešams lambdas simbols, lai spētu atšķirt lambdas no citām abstrakcijām.

Vienkāršākā funkcija, ar kuru pārbaudīt lambda sintaksi, ir identitātes funkcija un pseidokodā to var izteikt šādi:

```
func (x) { return x; }
```

Identitātes funkcija lambdas rēķinos tiek izteikta šādi:

```
λ x.x
```

Bet īsākā iespējamā identitātes funkcijas definīcija ir šāda:

```
x
```

Acīmredzami, vairs nav nekā, ko varētu vēl atmest, nezaudējot identitātes funkcijas semantiku, līdz ar to ir iegūts simbolu perfektas lambda izteiksmes piemērs. Un tas arī nozīmē, ka netieši parametri ļauj lambda sintaksi padarīt perfektu.

Programmēšanas valodas var uzturēt daudz dažādu abstrakciju, un ir loģiski, ka viena sintakse var nespēt apmierināt perfektus apstākļus visām uzturētajām abstrakcijām. Varētu būt, ka kāda programmēšanas valodas sintakse nevar tikt uzlabota, lai atbalstītu perfekto lambda sintaksi, neierobežojot atbalstu dažām citām abstrakcijām. Jebkurā gadījumā, visvienkāršākais veids kā implementēt perfekto lambda sintaksi ir radīt jaunu programmēšanas valodu bez

nepieciešamības raizēties par savietojamību ar vecajām programmatūras versijām un citiem ierobežojošiem faktoriem.

5 ABSTRAKTA PROGRAMMĒŠANAS VALODA APRĒĶINIEM

Netiešie parametri var tikt izmantoti jebkurā konceptā, kas var tikt interpretēts kā funkcija. Viens no šādiem konceptiem ir lambda – matemātiskas funkcijas abstrakcija un pamats lambdas aprēķiniem (Rojas R., 1998). Ja lambda rēķini tiek modificēti, ņemot vērā netiešos parametrus, rezultāts ir programmēšanas valoda vārdā „Kat" jeb „Kat-lang", kur identitātes funkcija var tikt deklarēta šādi ('fun1' dēvē par nosauktu funkciju):

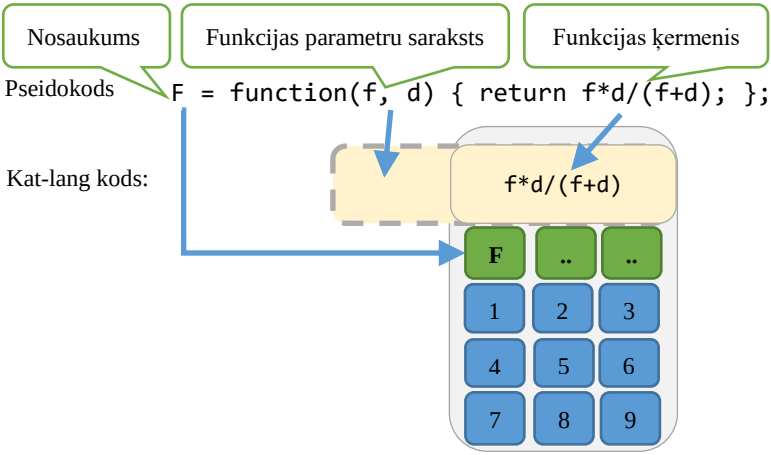
```
fun1 = x
```

Attiecīgi, konstrukcijas parametri tiek deklarēti konstrukcijas ķermenī, un nav nepieciešams speciāls simbols, lai atdalītu konstrukcijas parametru deklarāciju no konstrukcijas ķermeņa deklarācijas. Lietojot netiešos parametrus, visi nezināmie identifikatori, kas izmantoti konstrukcijas ķermenī, ir jāinterpretē kā konstrukcijas netiešie parametri. Tādējādi iepriekšējā koda fragmentā „x" ir netiešais parametrs, kas pēc definīcijas ir parametrs.

Kat-lang ir eksperimentāls funkcionālās programmēšanas valodas koncepts ar sintaksi, kas līdzīga vidusskolas algebras kursa sintaksei. Katrs, kurš zina algebru pamata līmenī, var veikt vienkāršus aprēķinus, izmantojot Kat-lang valodu. Papildu konstrukcijas kā nosacījuma parametri un ciklu konstrukcijas ir viegli apgūstamas. Tādējādi Kat-lang apguve būtu ātrāka par jebkuras standarta programmēšanas valodas, kas var tikt izmantota aprēķinu veikšanai, apguvi.

Vienkāršota Kat-lang valodas versija ir implementēta "IlCalculus" kalkulatora lietotnes prototipā (IlCalculus, 2015) Windows Phone platformai. "IlCalculus" kalkulators atšķiras no citām konkurējošām aplikācijām ar tā dizainu. Izmantojot netiešos parametrus funkciju definēšanā, ir iespējams atbrīvoties no uzlecošajiem logiem. Netiešo parametru inovācija ļauj lietotājiem lietot skārienjutīgā ekrāna iespēju "ievelc mani", lai definētu, labotu un veiktu jebkuru funkciju bez papildu uzlecošajiem ekrāniem (skat. **att. 5.1**). Spēja definēt un izpildīt jebkuru funkciju uz galvenā ekrāna ir īpaši svarīga mazekrāna ierīcēs.

Autors cer, ka Kat-lang valodas inovācijas nākotnē tiks lietotas, lai programmēšanu veiktu viedtālruņos.



att. 5.1. Funkciju definēšana viedtālruņa galvenajā ekrānā, lietojot netiešos parametrus un "ievelc mani" (drag&drop) iespējas izmantošana, lai izvairītos no uzlecošajiem ekrāniem.

6 STINGRI TIPIZĒTA METADATU APSTRĀDE

Tipu drošība ir tipu sistēmas svarīga īpašība. Modernajās programmēšanas valodās eksistē dažādi mehānismi kā darboties nezaudējot tipu drošību, piem., izmantot: īpašības, konstrukcijas, notikumus, atribūtus (anotācijas) un citas struktūras, bet nevienā no eksistējošajām vispārējās nozīmes programmēšanas valodām ar reflekcijas atbalstu nav elementu metadatu apstrādes mehānisma, kas pilnībā nodrošinātu tipu drošību. Eksistējošie risinājumi vispār nenodrošina tipu drošību vai nodrošina to tikai daļēji, turklāt eksistējošie risinājumi ir komplicēti, metadatu apstrāde notiek programmas izpildes brīdī un bieži tie nav programmēšanas valodas līmenī iestrādāti metadatu apstrādes mehānismi. Problēmu var risināt ieviešot jaunas metadatu apstrādes metodes.

Tipu metadatu piekļuves operators "typeof" (C#) un ".class" izsaukums (Java) rezultējas metadatu eksemplārā, kas satur apstrādātā tipa (klases, struktūras, saskarnes) metadatus. Līdzīgā veidā programmēšanas valodas var tikt papildinātas ar operatoru "memberof" tādā veidā, ka memberof(classField) rezultātā izdotu lauka metadatu FieldInfo eksemplāru (C#) vai Field (Java) eksemplāru. Microsoft ® korporācija bija pirmā, kas iepazīstināja ar ideju par elementu metadatu piekļuvi nezaudējot tipu drošību un viņi savu koncepciju nosauca par "infoof" (Lippert E., 2009). Microsoft aplūkoja operatoru "infoof" kā līdzīgu operatoram "typeof", kurš darbojas tikai ar tipu metadatiem:

```
Type info = typeof (int);
```

Operators “typeof” kā parametru pieņem tipu nevis eksemplāru. Atbilstoši valodas C# specifikācijai, šāds piemērs ir nederīgs:

```
Type x = typeof(6);
```

Elementa metadatu piekļuves operatora, kurš kā parametru nepieņem eksemplāra izteiksmes, lietošana ir pārāk specifiska un nav pietiekama, lai nodrošinātu tipu drošību elementu metadatu apstrādē.

Valodā C# 6.0 Microsoft ® iestrādāja vienkāršota operatora “infoof” versiju “nameof”, kas rezultātā izdod izteiksmes metadatu reprezentāciju simbolu virknē (C# 6.0, 2015):

```
var namespaceName = nameof(System.Collections.Generic);
//rezultējas simbolu virknes vērtībā: "Generic"
```

Operatora “nameof” pieejai ir 2 fundamentāli trūkumi:

1) Tas nepieņem konstrukciju parametru tipus un elementa rezultējošās vērtības tipu kā noskaņotos parametrus. Rezultātā, operatora “nameof” pielietošanas rezultātā iespējams zaudēt tipu drošību.
2) Kā rezultātu tas izdod tikai elementa nosaukumu, kas ir tikai maza daļa no visiem pieejamajiem elementa metadatiem, kas varētu tikt izdoti.

Abiem operatora “nameof” aplūkotajiem trūkumiem ir kopīga iezīme jeb cēlonis – operatora rezultējošās vērtības tips ir simbolu virknes datu tips nevis specializēts metadatu tips.

6.1 Operatora “memberof” izgudrošana priekš valodas C#

Šādi izskatās eksemplāra izveidošanas piemērs, kas tiks izmantos turpmākajos koda fragmentos:

```
var myFriend = new Person("Oscar");
```

Eksemplāra elementa metadatiem var piekļūt izmantojot norādi uz pašu eksemplāru:

```
FieldInfo metadata = memberof(myFriend.FullName);
```

Statiskā elementa metadatiem var piekļūt izmantojot tipa informāciju:

```
FieldInfo metadata = memberof(Person.TotalPersons);
```

Ietvarā .NET operators ‘memberof‘ varētu tikt pārlādēts šādās versijās:

- memberof(field) rezultātā jāizdod FieldInfo eksemplārs;
- memberof(method(parametru tipu saraksts – nav obligāts)) rezultātā jāizdod MethodInfo eksemplārs;
- memberof(property) rezultātā jāizdod PropertyInfo eksemplārs;
- memberof(class(parametru tipu saraksts – nav obligāts)) rezultātā jāizdod ConstructorInfo eksemplārs;
- memberof(event) rezultātā jāizdod EventInfo eksemplārs.

Ietvarā .NET klase MemberInfo ir bāzes klase šādām elementu metadatu klasēm: FieldInfo, MethodInfo, PropertyInfo, ConstructorInfo, EventInfo. MemberInfo eksemplāra lietošana ir daudz piemērotāka gadījumos, kad, saglabājot drošību, nepieciešams piekļūt tikai elementa nosaukumam bez piekļūšanas paplašinātai elementa informācijai.

Dažkārt nepieciešams piekļūt ne tikai elementa metadatiem, bet arī apstrādāt elementu ņemot vērā parametru vai parametrus, kuru tipiem jābūt savietojamiem ar elementu saturošā eksemplāra tipu. Piemēram, datu vaicājumos var būt noderīga šāda konstrukcija “FilterByEquality”:

```
public IEnumerable<object> FilterByEquality (MemberInfo
memberMetaData, object constrainedValue) {…}
```

Iepriekš aplūkotajā piemērā nav pilnībā saglabāta tipu drošība, jo parametra “constrainedValue” tips var nebūt savietojams ar elementa tipu uz kuru “memberMetaData” netiešā veidā norāda. Lai atrisinātu problēmas ar tipu drošību, promocijas darba autors piedāvā paplašināt elementu metadatu tipus ar noskaņotajiem parametriem. Piemēram, MemberInfo paplašināt līdz MemberInfo<TObject, TMember>, kur TMember norāda elementa tipu, bet TObject norāda eksemplāra tipu, kas satur apstrādājamo elementu.

Ietvarā .NET nepieciešams veikt šādas izmaiņas, lai iegūtu uzlabotā operatora “memberof” funcionalitāti:

- MemberInfo jāaizvieto ar MemberInfo<TObject, TMember>
- FieldInfo jāaizvieto ar FieldInfo<TObject, TMember>
- MethodInfo jāaizvieto ar MethodInfo<TObject, TMember>
- PropertyInfo jāaizvieto ar PropertyInfo<TObject, TMember>
- ConstructorInfo jāaizvieto ar ConstructorInfo<TObject, TMember>
- EventInfo jāaizvieto ar EventInfo<TObject, TMember>

Uzlabotā operatora “memberof” izmantošanas piemēri:

```
//piekļūšana elementa metadatiem zinot eksemplāru
var somebody = new Person("Anonymous");
MemberInfo<Person, string> memberMetadata =
    memberof(somebody.FullName);
```

```
//piekļūšana lauka metadatiem nezinot eksemplāru
FieldInfo<Person, string> fieldMetadata =
    memberof(Person.FullName);
```

6.2 Metadatu kombinēšana nodrošinot pilnīgu tipu drošību

Metadatu kombinēšana nozīmē klases elementu metadatu kombinēšanu ar datiem, tipu metadatiem un ierobežojumiem. Eksistējošie metadatu kombinēšanas risinājumi nodrošina tikai ierobežotu tipu drošību, tie ir sarežģīti, apstrāde var notikt programmas izpildes laikā (lēni), un tie nav valodā iebūvēti metadatu kombinēšanas risinājumi, tādēļ nespēj nodrošināt pilnīgu tipu drošību. Problēma var tikt atrisināta ieviešot jaunu sintaksi un metadatu kombinēšanas metodes tā, lai metadati tiktu apstrādāti kompilācijas brīdī un nodrošinātu pilnīgu tipu drošību (Vanags M., Cevere R., 2017).

Vairums moderno objektorientēto programmēšanas valodu ir stingri tipizētas (strongly typed) un daudzas no tām ļauj darboties ar reflekciju – metadatu piekļuves mehānismu.

Objektorientētās programmēšanas valodas, kuras ļauj darboties ar reflekciju, var tikt papildinātas ar metadatu kombinēšanas risinājumu nezaudējot tipu drošību. Tas ir panākams izmantojot operatoru "memberof" (Vanags M., et.al., 2013, Vanags M., et.al., 2014), kas ir operatora "nameof" uzlabota versija:

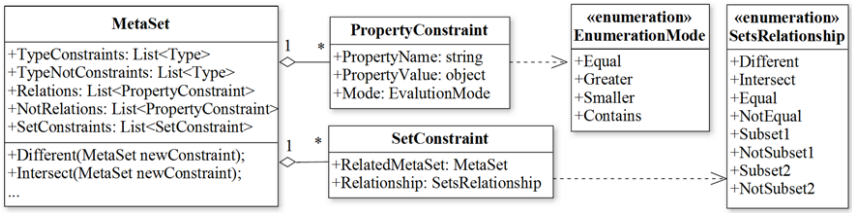
```
//Piekļūšana statiska lauka metadatiem ar operatoru "memberof"
FieldInfo<Person,string> metadata = memberof(somebody.FullName);
```

Lai iegūtu kodolīgāku sintaksi, metadatu piekļuves operators "memberof" var tikt pārsaukts par "meta". Tādā gadījumā metadatu piekļuves piemērs būs šāds:

```
//Piekļūšana statiska lauka metadatiem ar operatoru "meta":
FieldInfo<Person, string> metadata = meta(somebody.FullName);
```

Papildus metadatu tipu drošībai, dažkārt programmētājiem nepieciešams organizēt metadatus noteiktā veidā vai arī kombinēt metadatus ar datiem. Piemēram, lai glabātu datus relāciju datu bāzē, ir jāzina tabulas nosaukums, kolonnu nosaukumi (objektorientētajā pasaulē tie tiek saistīti ar klases laukiem) un vērtības. Domājot vispārīgāk – būtu noderīgi, ja eksistētu īpašības-vērtības (atslēgas-vērtības) abstrakcija, kas spētu ierobežot datus, lai tie varētu tikt saistīti ar konkrētu tabulas kolonu (relāciju datu bāzēs) vai datu struktūrām NoSQL datu avotos. Šāda īpašības-vērtības abstrakcija var tikt saukta par īpašības-ierobežojumu (Vanags M., et.al., 2014).

Relāciju datu bāzēs dati tiek glabāti tabulās, tādējādi datu vaicājuma rezultāts ir tabulas rindiņu kopa. Līdzīgi, NoSQL datu avotos datu (atslēgu-vērtību pāri) tiek glabāti strukturētā veidā, piem., objektos, grupās, u.c. struktūrās. Līdz ar to vispārīga datu vaicājuma abstrakcija var būt objektu kopas (līdzīgi kā tabulu rindiņu kopa) abstrakcija, kas tiek saukta par meta-kopu (Vanags M., et.al., 2014), (skat. **att. 6.1**).



att. 6.1. Vienkāršota meta-kopas fiziskā struktūra

Meta-kopas var tikt izmantotas datu vaicājumos. Spēja definēt meta-kopas vispārējās nozīmes programmēšanas valodās nozīmē abstraktas datu vaicājumu valodas integrēšanu vispārējās nozīmes programmēšanas valodās. Darbojoties ar datiem ir nepieciešama informācija par datu struktūrām un tie ir metadati. Datu izgūšanai no datu avota nepieciešami ne tikai metadati, bet arī izgūstamo datu definīcijas apgabals, jeb ierobežojumi. Tātad, datu vaicājumi nozīmē datu un metadatu kombinēšanu.

Darbā ar relāciju datu bāzēm jāzina apstrādājamo tabulu un kolonu nosaukumi. Darbā ar objektu datu avotiem jāzina objektu tipi (klases) un klašu lauki. Līdzīgas vajadzības eksistē manipulācijās ar citu veidu NoSQL datu avotiem, tādēļ datu vaicājumā nedalāma vienība ir īpašības-vērtības (atslēgas-vērtības) abstrakcija, kas tiek saukta par "īpašību-ierobežojumu". Relāciju datu bāzē, īpašības daļa īpašības-vērtības ierobežojumā norādīs uz nepieciešamo kolonu un vērtības daļa ierobežos izgūstamo vērtību (datu) definīcijas apgabalu. NoSQL datu avotos īpašības daļa īpašības-vērtības ierobežojumā norādīs uz nepieciešamo atslēgu (objektu datu bāzē tas būs klases lauks) un vērtības daļa ierobežos izgūstamo vērtību definīcijas apgabalu.

Klašu elementu metadati var tikt interpretēti kā īpašības daļa īpašību-ierobežojumos. Piemēram, ietvarā .NET, klases elementu metadatu tips ir klase MemberInfo. Lai kodolīgā veidā izveidotu īpašību-ierobežojumu eksemplārus, programmēšanas ietvaros darbā ar elementu metadatiem jābūt iespējai pārdefinēt relāciju operatorus: <, >, <>, ==, !=. Eksistējot uzlabotajam elementu metadatu tipam, paaugstinātos tipu drošība un īpašības-ierobežojumus varētu definēt šādi:

```
MemberInfo<Person, int> ageProperty = meta(Person.Age);
PropertyConstraint youngerThan30 = ageProperty < 30;
```

Īpašības ierobežojums var tikt definēts vienā rindā, turklāt, tā definēšanā, tipu iespējams deklarē netiešā veidā izmantojot operatoru "var":

```
var youngerThan30 = meta(Person.Age) < 30;
```

Kombinējot vienu vai vairākus īpašību ierobežojumus ar metadatiem (pamatinformāciju par datu struktūrām – tabulu nosaukumiem relāciju datu bāzēs, tipu nosaukumiem objektu datu bāzēs, utt.) rezultātā tiek iegūta jauna abstrakcija - datu vaicājumu abstrakcija "meta-kopa". Meta-kopa ir objektu kopas abstrakcija – tā apraksta datu ierakstus: rindas, objektus, u.c., atkarībā no datu avota arhitektūras. Meta-kopa nesatur objektus (datus), tā satur visu nepieciešamo informāciju, lai varētu uzģenerēt datu vaicājumu datu avotam, tādēļ meta-kopa var tikt interpretēta arī kā datu vaicājuma abstrakcija.

Meta-kopu fiziskā struktūra ir definēta pētījumā (Vanags M., et.al., 2014) un tā tiek izmantota kā pamats meta-kopu sintaksei:

```
MetaSet metaSetVariable = metaset
    [Type1    Selector1,    Selector2,...,    Type2    Selector3,
Selector4,...]
    [not TypeNot1, TypeNot2...]
    [where constr1, constr2...]
    [where not constrNot1, constrNot2...]
    [different metaDifferent1, metaDifferent2...]
    [intersect metaIntersect1, metaIntersect2...]
    [equal metaEqual1, metaEqual2...]
    [subset metaSubset1, metaSubset2...]
    [isSubsetOf metaIsSubsetOf1, metaIsSubsetOf2...]
    [notEqual metaNotEqual1, metaNotEqual2...]
    [notSubset metaSubset1, metaSubset2...]
    [isNotSubsetOf metaNotSubsetOf1, metaNotSubsetOf2...]
```

Kvadrātiskajās iekavās lietotais saturs nav obligāts.

7 META-KOPU RĒĶINU PAMATI

Eksistē dažādas ekspertu sistēmas un deduktīvās datu bāzu sistēmas, turklāt katrā tiek izmantota savādāka sintakse, kas būtiski atšķiras no moderno objektorientēto programmēšanas valodu sintakses. Ne tikai ekspertu sistēmu čaulas katra izmanto savādāku sintaksi, bet arī katra datu avota apstrādei var nākties izmantot pilnīgi savādāku sintaksi. Datu izgūšanā no dažādiem datu avotiem var palīdzēt abstrakcijas, piem., LINQ – vaicājumu atbalsts .NET valodās (LINQ, 2015). Bet LINQ nespēj palīdzēt, ja nepieciešams izmantot dedukciju, jo loģiskās secināšanas sistēma var paprasīt ielādēt atmiņā visu datu bāzes saturu, lai pārliecinātos, ka dedukcijas rezultāts ir korekts. Problēma var

tikt atrisināta, izmantojot objektu kopas abstrakciju – meta-kopu. Klasiskā predikātu loģika, kāda tā ir implementēta loģiskās programmēšanas valodā Prolog (Bratko I., 2000), nedarbojas ar meta-kopām. Lai varētu darboties ar meta-kopām, nepieciešams modificēt loģiskās programmēšanas dzini un šādas modifikācijas noved pie otrās kārtas predikātu loģikas paveida.

Šajā pētījumā pirmās kārtas predikātu loģika tiek salīdzināta ar otrās kārtas predikātu loģikas paveidu, kas darbojas ar meta-kopu rēķiniem. Tas ir galvenais iemesls, kādēļ pētījumā netiek tiešā veidā salīdzināti meta-kopu rēķini ar Prolog, DataLog (Bancilhon F., et.al., 1986) un citām sistēmām, kuras izmanto pirmās kārtas predikātu loģiku. Sistēmas, kuras izmanto pirmās kārtas predikātu loģiku, var izmantot dažādus kopu simulēšanas paņēmienus, piem., izmantot sarakstus, bet saraksti nav matemātiskas abstrakcijas. Saraksti ir datu struktūras, kuras var saturēt galīgu skaitu elementu, bet tajā pat laikā kopas var saturēt nezināmu skaitu elementu vai pat būt bezgalīgas.

Abstraktāka domāšana ļauj izveidot labāku dizainu (Pierce B.C., 2002). Eksistē daudz dažādas abstrakcijas (saskarnes, mantošana, tipu sistēmas, u.c.), loģikas (kombinatorā loģika, izteikumu loģika, predikātu loģika, u.c.), matemātiskie rēķini (piem., lambda rēķini, kas paplašina kombinatoro loģiku ar matemātiskās funkcijas abstrakciju (Душкин Р.В., 2006)), pieejas (piem., MDA (Model Driven Architecture (MDA, 2013)). MDA definē tikai aksiomātiskos likumus un nosaka modeļu un meta-modeļu aprakstošo sintaksi, bet transformāciju loģika netiek definēta, katrs programmētājs to definē atšķirīgi. Līdz ar to šāda pieeja nevar būt viennozīmīgi interpretējama. Abstrakciju piemēri ar meta, meta-meta un tamlīdzīgiem modeļiem noved pie jautājuma: "Kā klasificēt sistēmu, kura darbojas ar meta-kopām?" Tā kā meta-kopu sistēmas pamatā izmanto predikātu loģiku, tad tā nav pieeja (kā MDA), meta kopu sistēma ir vismaz loģika vai matemātiskie rēķini. Formālajos rēķinos pierakstītas izteiksmes var tikt pārveidotas uz formālo rēķinu pamatā esošo loģiku (piemēram, lambda rēķinus iespējams pārveidot kombinatorajā loģikā un otrādi). Meta-kopu sistēma ir balstīta uz predikātu loģiku, bet meta-kopu apstrādes operācijas neeksistē tradicionālajā predikātu loģikā un meta-kopas nevar pārveidot uz pirmās kārtas predikātu loģiku. Pirmās kārtas predikātu loģika izmanto mainīgos, kas var tikt aizvietoti ar objektiem, bet otrās kārtas predikātu loģika izmanto mainīgos, kas var tikt aizvietoti gan ar objektiem, gan ar objektu kopām (Second-order logic, 2017; Cohen M.S., 2004).

Meta-kopa ir objektu kopas abstrakcija un meta-kopu rēķinu pamatā ir otrās kārtas predikātu loģika, kur objektu vietā tiek izmantotas meta-kopas. Meta-kopas nesatur reālos objektus un tās nesatur arī norādes uz reālajiem objektiem; meta-kopas var saturēt tikai ierobežojumus, kurus iespējams izmantot, lai ģenerētu vaicājumus datu avotiem un izgūtu reālos objektus. Meta-kopu rēķinu dzinis var tikt implementēts uzlabojot loģiskās programmēšanas dzini, lai tas spētu darboties ar meta-kopām (Vanags M., et.al., 2014). Nākamajās nodaļās tiks dots meta-kopu detalizēts apraksts un skaidrojums.

Atšķirība starp predikātu loģiku un meta-kopu rēķiniem ir veidā, kādā fakti (biznesa objekti) tiek apstrādāti. Darbojoties ar faktiem tiešā veidā, programmēšanas dzinis var paprasīt ielādēt atmiņā visu datu bāzes saturu, lai spētu sekmīgi pabeigt deduktīvās secināšanas procesu. Šī problēma ir atrisināta meta-kopu programmēšanas dzinī, pateicoties meta-kopas abstrakcijai. Viena meta-kopa spēj attiekties uz daudz faktiem, līdz ar to meta-kopu programmēšanas dzinī izmantoto meta-kopu skaits būs daudz mazāks nekā faktu skaits datu bāzē. Līdz ar to meta-kopu programmēšanas dzinis darbosies daudz ātrāk un efektīvāk, turklāt tas būs neatkarīgs no biznesa objektu datu bāzes satura, tādējādi padarot dedukcijas procesu decentralizētu.

Meta-kopu rēķini ir līdzīgi ierobežojumu loģikas programmēšanai (constraint logic programming) (Jaffar J., Maher M.J., 2013), tikai meta-kopu rēķini papildus spēj apstrādāt meta-kopu abstrakcijas un informāciju par objektu tipiem.

7.1 Meta-kopas loģiskā struktūra

Katra meta-kopa var sastāvēt no 3 dažādām ierobežojumu kopām: tipu ierobežojumiem, īpašību ierobežojumiem un kopu ierobežojumiem. Ieteicamā meta-kopu sintakse priekš meta-kopu rēķiniem ir šāda:

```
<tipu ierobežojumi> [īpašību ierobežojumi] {kopu ierobežojumi} |
substitūcijā iesaistītais mainīgais
```

Obligātā meta-kopu sintakses konstrukcija ir tipu ierobežojumi, kas vienmēr tiek deklarēti pirms visiem citiem meta-kopu ierobežojumiem. Īpašību ierobežojumi un kopu ierobežojumi nav obligāti un tos var nedefinēt. Katrs ierobežojums var tikt deklarēts kopā ar NOT operatoru, taču NOT operators nemaina ierobežojuma tipu. Ja meta-kopa satur vairāk par vienu viena veida ierobežojumu, tad ierobežojumi to deklarācijā tiek atdalīti ar komatu. Substitūcijā iesaistītais mainīgais nav obligāts un tas tiek lietots tikai, lai attēlotu meta-kopu apstrādes operācijas dedukcijas procesā. Substitūcijā iesaistītais mainīgais netiek glabāts datu bāzē, kā arī netiek izmantots, lai ģenerētu vaicājumu datu avotam.

Tipu ierobežojumi tiek lietoti, lai specificētu objektu tipus objektu kopai, kuru reprezentē meta-kopa. Ja tipu ierobežojumi tiek lietoti kopā ar NOT operatoru, tad rezultējošais ierobežojums reprezentē objektu tipus, ko meta-kopas reprezentētā objektu kopa nesaturēs. Meta-kopu rēķini ir veidoti lietošanai objektorientētā vidē un meta-kopu tipu ierobežojumi ir savietojami ar tipu mantošanu, saskarnes (interface) implementēšanu un citām objektorientētās vides abstrakcijām.

Meta-kopas piemērs, kur meta-kopa reprezentē klases (tipa) Animal objektus, kas nav klases Dog objekti:

```
<Animal, not(Dog)>
```

No iepriekšējā piemēra iespējams uzģenerēt datu bāzes vaicājumu, kura rezultāts saturētu visus klases Cat un Giraffe objektus, jo šie tipi manto klasi Animal, bet nemanto klasi Dog.

Lietojot NOT operatoru tipu ierobežojumu definēšanā, ieteicams meta-kopas deklarācijā norādīt vismaz vienu tipu ierobežojumu, kuram netiek piemērots NOT operators, jo vaicājumu konstruēšana dažādās datu bāzu sistēmās ir implementēta dažādi un tipu ierobežojumi tikai ar NOT operatoriem samazina viennozīmīgas vaicājuma interpretācijas iespējas un var novest pie neprognozējamiem rezultātiem (piem., ja netiek norādīti tipu ierobežojumi bez NOT operatora, tad objektu datu bāze Db4o (Paterson J., Edlich S., 2006) var ignorēt pielietoto NOT operatoru). Objektorientētās vidēs ir ērti izmantot saskarnes (interface), lai ierobežotu meta-kopas reprezentēto objektu tipus.

Īpašību ierobežojumi ļauj definēt vēlamo vērtību kopu konkrētām īpašībām. Ieteicamā īpašību ierobežojumu sintakse ir šāda:

```
[ĪpašībasNosaukums SalīdzināšanasRežīms IerobežotāVērtība]
```

Meta-kopu rēķinu pamatā ir 4 dažādi īpašību vērtību salīdzināšanas režīmi: vienāds (Equal), lielāks (Greater), mazāks (Smaller) un satur (Contains). Nākamais piemērs demonstrē meta-kopu, kas reprezentē nedresētus suņus, jaunākus par 3 gadiem:

```
<Dog>[Age<3,not(Trained=True)]
```

Īpašību ierobežojumu vērtības var būt arī meta-kopas, kas var saturēt savus tipu, īpašību un pat kopu ierobežojumus. Nākamais piemērs ilustrē, kā iespējams deklarēt kompleksu meta-kopu, kas reprezentē personas, kurām pieder vismaz viens dresēts suns:

```
<Person>[Pets contains <Dog>[_trained=True]]
```

Kopu ierobežojumi ir unikāla darbošanās ar otrās kārtas predikātu loģiku. Izmantojot kopu ierobežojumus, ir iespējams definēt attiecības starp divām dažādām meta-kopām. Meta-kopa var tikt interpretēta kā datu bāzes vaicājums, kura rezultātā tiek iegūti reālie objekti (dati). Kopu ierobežojumu definēšanas sintakse ir šāda:

```
{KopuIerobežojumaVeids(meta-kopa)}
```

Kopu ierobežojumi vienmēr tiek definēti kā attiecība starp divām meta-kopām: pirmā ir meta-kopa (konteksta meta-kopa), kas satur definējamo kopu ierobežojumu, bet otrā meta-kopa tiek izmantota kā ierobežojuma vērtību apgabals priekš konteksta meta-kopas. Gadījumos, kad nepieciešams definēt

attiecības starp vairāk kā divām meta-kopām, var izmantot vairākus kopu ierobežojumus.

Eksistē 8 dažādi kopu ierobežojumu veidi: dažādas (Different), šķeļas (Intersect), vienādas (Equal), nav vienādas (NotEqual), pirmā kopa ir apakškopa otrajai (Subset1), pirmā kopa nav apakškopa otrajai (NotSubset1), otrā kopa ir apakškopa pirmajai (Subset2), otrā kopa nav apakškopa pirmajai (NotSubset2).

Nākamais piemērs ilustrē, kā, izmantojot kopu ierobežojumu, iespējams definēt meta-kopu, kas reprezentē visus suņus, kuri nav trenēti:

```
<Dog>{different(<Dog>[Trained=True])}
```

Iepriekšējo meta-kopu SQL vaicājuma formā var izteikt šādi:

```
SELECT * FROM Dogs
EXCEPT
SELECT * FROM Dogs WHERE Trained='True';
```

7.2 Meta-kopu sakrišanas operācija un unifikācija

Loģiskās programmēšanas dzinis darbojas ar objektiem, bet to iespējams modificēt, lai tas varētu darboties arī ar meta-kopām. Šādā gadījumā predikātu termi var būt meta-kopas:

```
dog(<Dog>).
trained(<Pet>[Trained=True]).
```

Tā kā meta-kopas nemaina sintaksi izmantotajiem mainīgajiem, to lietojums neietekmē likumu deklarācijas sintaksi. Piemērs, kurā, izmantojot meta-kopas, tiek deklarēts fakts predikātu loģikā:

```
trainedDog(x) :- dog(x), trained(x).
```

Loģiskajā programmēšanā sintaktiski identiski objekti sakrīt, bet sintaktiski nesakrīt. Sakrišanas operācijā nav nozīmes vai tiek salīdzināti atomi vai termi. Svarīga ir atšķirība: vai tiek salīdzināti mainīgie vai struktūras, kas nav mainīgie:

```
female(Ineta) sakrīt ar female(Ineta)
female(Ineta) sakrīt female(x) mainīgais x var tikt aizvietots
ar objektu Ineta
female(Ineta) nesakrīt ar female(Alice).
```

Meta-kopu sakrišanas operācija atšķiras no objektu sakrišanas, jo meta-kopas ir kā mazas daļas no lielāka vaicājuma, kurš atrodas konstruēšanas

procesā, tādēļ ne visas atšķirības meta-kopās tiek interpretētas kā nesekmīgas sakrišanas. Sakrišana nesaista mainīgos ar meta-kopām, to dara unifikācija.

Ja meta-kopu tipu ierobežojumi unifikācijas procesā sakrīt un, ja unifikācijā tiek izmantots mainīgais, tad meta-kopa, uz kuru norāda mainīgais, saturēs tipu, īpašību un kopu ierobežojumus no abām unifikācijas procesā iesaistītajām meta-kopām, kā arī mainīgo, kurš tika iesaistīts unifikācijas procesā. Piemēram, ja zināšanu bāzē atrodas šāds fakts un likums:

```
dog(<Dog>). trained(<Pet>[Trained=True]).
trainedDog(x) :- dog(x), trained(x).
```

Tad uzdodot meta-kopu rēķinu dzinim jautājumu trainedDog(n)?, dzinis veiks unifikāciju ar termiem trainedDog(n) un trainedDog(x), kā rezultātā mainīgais x tiks aizstāts ar mainīgo n un tiks izveidots jauns mērķu saraksts: dog(n) un trained(n). Dog(n) sakritīs ar zināšanu bāzes faktu dog(<Dog>), kā rezultātā mainīgais n tiks saistīts ar meta-kopas <Dog>|n kopiju. Līdz ar to, mērķis tiks atjaunināts uz šādu stāvokli: trained(<Dog>|n). Trained(<Dog>|n) sakrīt ar trained(<Pet>[Trained=True]), rezultātā mainīgais n tiek saistīts ar šādu meta-kopu: <Dog>[Trained=True]|n. Mērķu saraksts ir tukšs un mainīgais n norāda uz meta-kopu <Dog>[Trained=True]|n, kas arī ir atbilde uz sistēmai uzdoto jautājumu un mainīgais n vairs nav nepieciešams.

Unifikācija izmantojot NOT operatoru darbojas līdzīgi, vienīgā atšķirība ir tā, ka ierobežojumi, uz kuriem attiecas NOT operators tiek marķēti ar NOT marķieriem, bet ierobežojumi, kuri jau bija marķēti ar NOT marķieriem, tiks atbrīvoti no tiem. Piemēram, ja zināšanu bāze satur likumu: notTrainedDog:

```
notTrainedDog(x) :- dog(x), not(trained(x)).
```

Tad meta-kopu rēķinu dzinis uz jautājumu notTrainedDog(n)? sniegs šādu atbildi:

```
<Dog>[not(Trained=True)]|n.
```

7.3 NOT operators meta-kopām

Loģiskajā programmēšanā izteiksme NOT(P) nosaka, vai teorēma P atrodas zināšanu bāzē vai nē. Operators NOT pārtrauc atbildes meklēšanu tiklīdz tiek atrasts pirmais pierādījums teorēmas P eksistencei. Taču šāda uzvedība nav pareiza meta-kopu rēķinos, jo meta-kopas nav reāli objekti, meta-kopas ir tikai metadati un fakts NOT(P) meta-kopu rēķinos ir jāispēj saglabāt meta-kopā P kā ierobežojums. Meta-kopu rēķinos nepieciešams veikt meklēšanu zināšanu bāzē, lai pārbaudītu visus NOT argumenta pierādījumus (nepietiek, ja tiks atrasts tikai

pirmais teorēmas P pierādījums) – tikai tā iespējams iegūt pareizus NOT ierobežojumus priekš visām dedukcijas procesā izmantotajām meta-kopām.

NOT operators var tikt lietots kopā ar visiem iepriekš apskatītajiem operatoriem, izņemot operatoru "is implied by". Ja NOT operators tiek lietots kopā ar AND vai OR operatoriem, tad rezultējošo izteiksmju vienkāršošanā tiek izmantoti De Morgāna likumi.

7.4 Vairāki vaicājumi vienā jautājumā meta-kopu dzinim

Visos iepriekš aplūkotajos jautājumos tika izmantots tikai viens mainīgais, bet loģiskā programmēšana ļauj uzdot jautājumus, kas satur vairāk par vienu mainīgo:

```
father(x, y)?
```

Šādā gadījumā loģiskās programmēšanas dzinis atgriezīs visus tēva-dēla objektu pārus šādā formā:

```
x=Father1, y=Son1;
x=Father2, y=Son2.
…
```

Meta-kopu rēķinos arī iespējams veidot jautājumus, kas satur vairākus mainīgos. Taču, tā kā meta-kopas ir abstrakcijas, kas reprezentē datu bāzu vaicājumus, tad gadījumos, kad jautājumi satur vairākus mainīgos, meta-kopu rēķinu dzinis izpildīs tik daudz datu izgūšanas vaicājumus, cik daudz mainīgo būs jautājumā. Jāņem vērā, ka meta-kopas pēc definīcijas ir kopas un kopās objektu secība nav noteikta. Tas nozīmē, ka veidojot jautājumus ar vairākiem mainīgajiem, piem., "tēviem" un "dēliem", pēc vaicājumu izpildes un datu izgūšanas no datu bāzes, "tēvu" un "dēlu" secība var nesakrist.

8 ABSTRAKTA DATU IZGŪŠANAS TEHNOLOĢIJA

Nodaļu saturs, kurās aplūkoti meta-kopu rēķini, ir atkarīgs no veiktajiem eksperimentiem. Eksperimenti tika veikti izmantojot .NET platformu un programmēšanas valodu C# (C# 5.0, 2012; Skeet J., 2013), jo tieši valoda C# kopā ar objektu datu bāzi Db4o db4o (Paterson J., Edlich S., 2006) iedvesmoja autoru atklāt meta-kopu rēķinus un tos implementēt Deduktīvā Dedukcijas Dziņa (DDE) formā.

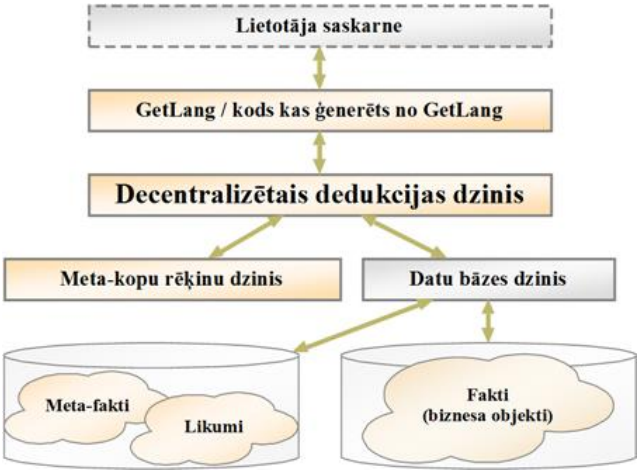
att. 6.1 parādīta meta-kopas fiziskā struktūra. Tipu ierobežojumi (TypeConstraints) ir saraksts, kas satur tipu ierobežojumus, bet TypeNotConstraints saraksts satur tipu ierobežojumus, kuriem piemērots NOT

operators. Īpašību ierobežojumi ir meta-kopu rēķinu pamatelements. Ja biznesa objektu datu bāze satur 1000 personas un 200 no tām ir jaunākas par 20 gadiem, tad visas 200 personas (fakti) var tikt saistītas vienā meta-kopā, kas satur tikai vienu īpašības ierobežojumu: "Age < 20". Līdzīgā veidā tiek apstrādāta kolekcija NotRelation jeb īpašību ierobežojumu saraksts, kurus nevar attiecināt uz vēlamo objektu kopu.

Konstrukcijas "GetVariable" un "SetVariable" tiek izmantotas īslaicīgai substitūcijas rezultātā aizvietoto mainīgo glabāšanai. Aizvietotie mainīgie ir svarīga informācija, kas tiek izmantota dedukcijas procesā, bet netiek glabāta datu bāzē, tādēļ meta-kopas implementācijā priekš objektu datu bāzes Db4o, lauks "variable" tiek marķēts ar atribūtu "[Transient]" (tas nosaka, ka lauks netiks glabāts datu bāzē).

Klase "Type" tiek izmantota, lai vispārinātu tipu ierobežojumu pieeju. Lai varētu saglabāt datus datu bāzē, vairumā gadījumu pietiek ar klases "Type" reprezentāciju "string" tipa formā. Piemēram, Db4o datu bāzes gadījumā, tipu saglabāšana datu bāzē nozīmē speciālu objektu translatoru izmantošanu, kas pārveido "Type" objektus uz "string" objektiem un otrādi. Līdzīgas problēmas ir arī ar "CultureInfo" un citiem tipiem, kas atkarīgi no platformas .NET kodola uzvedības un ir sarežģīti glabājami.

DDE arhitektūra parādīta **att. 8.1**, kur redzama mijiedarbība ar meta-kopu rēķinu dzini un datu glabāšanas sistēmu, lai izveidotu strukturētu datu deduktīvo datu bāzi.



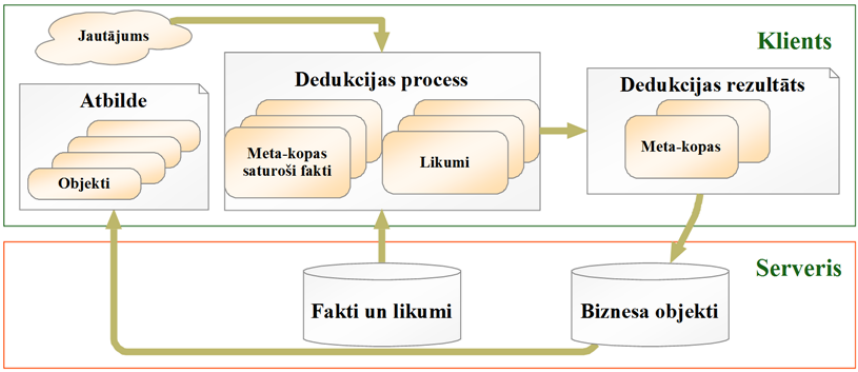
att. 8.1. Decentralizētā dedukcijas dziņa arhitektūra

Šādas arhitektūras priekšrocība ir biznesa objektu nodalīšana no meta-kopu (meta-faktu) un likumu datu bāzes (likumi ir līdzīgi meta-faktiem ar

vienīgo atšķirību – meta-fakti raksturo faktus, bet likumi kombinē kopā meta-faktus). Šāda nodalīšana uzlabo meta-faktu atkārtotās izmantošanas iespējas un ļauj datus glabāt centralizēti, ar dedukciju saistītu informāciju glabāt uz klienta datoriem. Turklāt, katram lietotājam (klientam) meta-fakti un likumi var atšķirties atkarībā no viņa vajadzībām.

8.1 Decentralizētais dedukcijas dzinis

Decentralizētais dedukcijas dzinis jeb DDD apvieno metakopu loģisko programmēšanu ar datu bāzes vaicājumu sistēmu un ļauj citiem komponentiem (piem., lietotāja saskarnes slānim) ar to mijiedarboties. DDD darbības shēma parādīta **att. 8.2**. Dedukcijas procesa rezultāts ir meta-kopu kolekcija. Ja datu bāze satur vairāk par vienu katra tipa objektu (relāciju datu bāzu gadījumā tas nozīmētu, ka katrā tabulā ir vairāk par vienu rindiņu), tad rezultējošo meta-kopu skaits būs ievērojami mazāks par datu bāzē esošajiem objektiem. Tas nozīmē, ka loģiskā programmēšana ar meta-kopām būs daudz ātrāka nekā ar objektiem. Meta-kopas struktūra ir līdzīga SODA (SODA, 2002) vaicājumu struktūrai, kas tiek izmantota objektu datu bāzēs, tādējādi meta-kopu kolekcija var tikt izmantota, lai automātiski uzģenerētu SODA vaicājumus datu bāzei un no tās izgūtu biznesa objektus. Kopā tas viss nozīmē, ka, lai spētu darboties korekti, DDD nav nepieciešams ielādēt atmiņā pilnīgi visu datu bāzes saturu.



att. 8.2. Konceptuāla decentralizētās dedukcijas dziņa darbības shēma

Viens no projekta mērķiem bija integrēt meta-kopu rēķinu sintaksi eksistējošajās programmēšanas valodās. Praktiskie eksperimenti vairāk koncentrējās uz .NET platformu un programmēšanas valodu C#, taču līdzīgā veidā meta-kopu rēķinu sintaksi var pielāgot arī citām programmēšanas valodām.

Jautājuma apstrādes darbplūsma decentralizētajā dedukcijas dzinī parādīta **att. 8.3** demonstrējot divu dažādu sintakses formu apstrādi.

att. 8.3. Jautājuma apstrādes darbplūsma decentralizētajā dedukcijas dzinī

Termu objekti var tikt saglabāti datu bāzē un tie ir platform neatkarīgi, tādēļ risinātājs spēj darboties tikai ar termu objektiem pārvērstiem normālformā. DDE veic normalizācijas pārveidojumus un sagatavo datu struktūras risinātājam vajadzīgajā formā.

Ideja par Prolog tipa sintaksi programmēšanas valodā C# pirmo reizi tika publicēta Microsoft LINQ CTP versijā 2006. gadā kā loģiskās programmēšanas piemērs (Orcas, 2006). LINQ projekts kļuva par .NET 3.5 sastāvdaļu, bet loģiskās programmēšanas piemēra projekts klusām tika izņemts no .NET materiālu klāsta. Promocijas darba autors uzskata, ka tas notika nevis tādēļ, ka Microsoft mēģinātu slēpt kādas potenciāli vērtīgas idejas, bet gan tādēļ, ka projekts bija pārāk sarežģīts priekš "vidējā programmētāja" un Prolog tipa sintaksei bez meta-kopu rēķiniem ir mazas pielietojuma iespējas vispārējās nozīmes programmēšanas valodās. Lai spētu darboties pareizi, loģiskās programmēšanas dzinim jāspēj piekļūt visam datu bāzes saturam un sliktākajā gadījumā tas var mēģināt ielādēt atmiņā visu datu bāzes saturu. Turpretī meta-kopu rēķinu dzinim nav nepieciešama piekļuve datu bāzes saturam, lai tas spētu darboties pareizi.

8.2 Meta-kopu rēķinu dzinis

Meta-kopu rēķinu dzinis darbojas tikai ar TermNode objektiem normalizētiem konjunktīvajā normālformā. Lielāka prioritāte par normalizācijas procesu ir NOT operatora apstrādei, kas izmanto De Morgāna likumu un citus pārveidojumus.

Meta-kopu rēķinu dziņa pamatā ir modificēts unifikācijas algoritms, kas kombinē meta-kopu ierobežojumus kā aprakstīts nodaļā 7.2. Modificētais unifikācijas algoritms spēj darboties gan ar meta-kopām, gan ar biznesa objektiem, taču šāda hibrīda uzvedība ir kompleksa un atsevišķu niansu darbināšanai var būt nepieciešami papildu pētījumi. Unifikācijas algoritma uzlabošana var notikt 2 galvenajos virzienos: 1) unifikācija, kas darbojas tikai ar meta-kopām; 2) hibrīdā unifikācija, kas darbojas gan ar meta-kopām, gan ar biznesa objektiem. Meta-kopu unifikācija ir ērta datu izgūšanā un dedukcijas procesā, bet hibrīdā unifikācija var būt vairāk noderīga ekspertu sistēmās.

Not operatora implementēšana meta-kopu dzinī ir vienkāršāka nekā skaidrots meta-kopu rēķinu pamatu nodaļā. Fiziskais meta-kopas modelis satur divus ierobežojumu sarakstus (skat Fig 2): viens saraksts priekš tipu ierobežojumiem bez NOT operatora un otrs saraksts priekš tipu ierobežojumiem ar NOT operatoru. Kad meta-kopu rēķinu dzinis pamana meta-kopas unificēšanu ar meta-kopu no zināšanu bāzes predikāta, tipu ierobežojumi no pirmā predikāta tiek pievienoti otrā predikāta tipu ierobežojumu sarakstam. Ja meta-kopu dzinis pamana NOT predikātu, tad saturs no pirmās meta-kopas tipu ierobežojumu saraksta tiek pievienots otrās meta-kopas TypeNotConstraints sarakstam. Līdzīgā veidā tiek apstrādāti īpašību ierobežojumi. Divi ierobežojumu saraksti ir ērti dziņa paplašināšanai ar nākotnes funkcionalitāti. Piemēram, jaunu ierobežojumu izgudrošana neprasītu izmaiņas visā ierobežojumu apstrādes sistēmā un jaunie ierobežojumi spētu darboties kopā ar NOT operatoru bez papildus izmaiņām NOT operatora darbībā. Vienīgi kopu ierobežojumi tiek glabāti vienā sarakstā, jo kopu ierobežojumi ir daudz sarežģītāki. Kopu ierobežojumi lietoti kopā ar NOT operatoru nozīmē kopu ierobežojuma SetsRelationship vērtības izmaiņu uz pretēju vērtību.

Iebūvēto predikātu apstrāde notiek meta-kopu rēķinu dzinī dedukcijas procesa laikā, tādēļ NOT operators un iebūvētie predikāti: IsTypeOf, Greater, Smaller, EqualTo, Different, Equal, Subset1, Subset2 un Intersect konstruē rezultējošo meta-kopu sarakstu dinamiski. Ja iespējams, tad ieteicams faktus deklarēt, izmantojot pilnu meta-kopas specifikāciju nevis konstruēt meta-kopas dinamiski, jo dinamiska meta-kopu konstruēšana ir nedaudz lēnāka un mazāk tipu droša operācija.

8.3 Datu vaicājumu ģenerēšana no meta-kopu saraksta

Datu bāzes vaicājuma ģenerēšana ir DDE otra svarīgākā komponente pēc meta-kopu rēķinu dziņa. Pašreizējā DDE implementācijā ir realizēta tikai SODA vaicājumu ģenerēšana priekš datu bāzes db4o. Nākamais piemērs demonstrē, kā tiek ģenerēts vaicājums no dedukcijas rezultātā iegūtas meta-kopas, ja zināšanu bāze satur šādus faktus:

```
dog(<Dog>).
trained(<Pet>[Trained=True]).
person(<Person>).
```

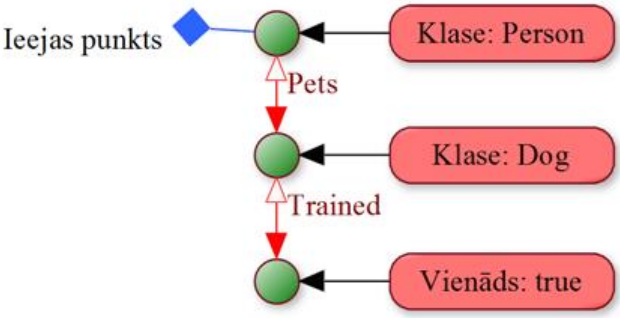
Un šādu likumu:

```
personWhoOwnsDog(x) :- person(x), dog(y), trained(y),
contains(x, "Pets", y).
```

Tad uzdodot sistēmai jautājumu "personWhoOwnsDog(n)?", tiks iegūta šāda atbilde meta-kopas formā:

```
<Person>[Pets contains <Dog>[Trained=True]]|n.
```

Pēc tam no meta-kopas automātiski tiek ģenerēts SODA vaicājums (vizuāli vaicājums parādīts **att. 8.4**). Uzģenerētais vaicājums tiek izpildīts un rezultātā iegūtie objekti tiek atgriezti lietotājam kā atbilde uz lietotāja uzdoto jautājumu DDE sistēmai.



att. 8.4. SODA vaicājuma ģenerēšana no meta-kopas "personWhoOwnsDog"

Pašreizējā DDE implementācija atbalsta vaicājumu ģenerēšanu datu bāzei db4o. Līdzīgā veidā iespējams paplašināt vaicājumu ģenerēšanas iespējas priekš citiem datu avotiem, kas spējīgi glabāt atslēgas-vērtības pārus strukturētā formā, piem., XML faili. Iespējams implementēt vaicājumu ģenerēšanas (no meta-kopām) atbalstu arī relāciju datu bāzēm, taču šajā jomā nepieciešams veikt papildus pētījumus, jo SQL vaicājumi var saturēt JOIN operācijas, kuras ir sarežģīti interpretēt objektorientētajā vidē. Join operators ļauj kombinēt kopā dažādas daļas no vairākām tabulām, bet objektorientētā pasaule sagaida, ka no datu bāzes tiks ielādēti veseli objekti nevis tikai objektu sastāvdaļas. Šo problēmu iespējams risināt interpretējot vaicājumus ar JOIN operatoru kā pieprasījumus ģenerēt rezultējošos objektus dinamiski un aizpildīt ar saturu, kas izgūts no datu bāzes, taču tam nepieciešami padziļināti pētījumi.

8.4 Vaicājumu valoda bez netiešajiem parametriem un netiešajiem argumentiem

"Invoice" objekts satur īpašības "Warehoue" un "DealDate". "Warehouse" ir lietotāja definēts tips, bet "DateTime" ir sistēmas tiks, kas reprezentē laiku. Komats likumu deklarācijās darbojas kā loģiskais UN operators, semikols nozīmē teikuma beigas, simbols ":-" nozīmē operatoru "is implied by" un simboli "==", ">=" un "<" reprezentē salīdzināšanas operācijas, kas meta-kopu kontekstā darbojas kā ierobežojumus veidojošās konstrukcijas. Meta-kopas ir objektu kopu abstrakcijas un tās jāinterpretē kā mazi vaicājumi, kurus, kombinējot kopā, veidojas komplicētāks vaicājums. Meta-kopas (ar meta-kopu mainīgajiem) ir nepieciešamas, lai nodrošinātu unifikācijas procesa korektu darbību, tādēļ meta-kopu mainīgos nevar interpretēt kā netiešos parametrus vai netiešos argumentus (tie vienmēr ir tieši; teorētiski iespējams tos padarīt netiešus, bet tas atstātu negatīvas sekas uz koda lasāmību, jo pārāk daudz svarīgas informācijas būtu ļauts deklarēt netiešā veidā). Kad meta-kopa tiek izmantota likuma sasniedzamības zonā, tās mainīgā nosaukums tiks pieņemts kā meta-kopas-mainīgais + automātiski-ģenerēt- skaitlis (šis process notiek fonā un patiesais meta-kopu mainīgā nosaukums programmētājam nav zināms).

Piemēram, šādu GetLang kodu:

```
metaset i as Invoice;
atWarehouse(i) :- i.Warehouse == storage;
inPeriod(i) :- i.DealDate>=startDate, i.DealDate<endDate;
buyingAtWarehouseInPeriod(i) :- atWarehouse(i), inPeriod(i);
```

kompilators interpretēs šādi:

```
metaset i as Invoice;
atWarehouse(i1) :- i1.Warehouse == storage;
inPeriod(i2):-i2.DealDate>=startDate,i2.DealDate<endDate;
buyingAtWarehouseInPeriod(i3) :- atWarehouse(i3), inPeriod(i3);
```

Tādējādi iespējams sasniegt kodolīgu sintaksi un nodrošināt visus priekšnosacījumus, lai unifikācijas process varētu noritēt korekti. Netiešo parametru un netiešo argumentu ideja ir neatkarīga no unifikācijas procesa, jo, pirms unifikācijas procesa, visi netiešie elementi vienalga tiek pārvērsti par tiešajiem elementiem. Sintaktiskā forma: someVariable.SomeProperty tiek lietota, lai piekļūtu objekta "someVariable" īpašībai "SomeProperty". Valodas semantika var tikt aplūkota kā vaicājuma konstruēšanas process. Loģiskās programmēšanas dzinis darbojas ar meta-kopu abstrakcijām (kombinējot kopā dažādus ierobežojumus), tādējādi rezultātā tiek iegūts meta-kopu saraksts jeb saraksts no abstrakcijām. Lai izgūtu datus (biznesa objektus) no datu bāzes, no iegūtā meta-kopu saraksta ir jāģenerē vaicājums vai pat vairāki vaicājumi datu

bāzei. Vaicājumu ģenerēšanas process nepieciešams, lai GetLang valodas kodu varētu pārveidot par kodu citās programmēšanas valodās.

```
//meta-kopas deklarācija
metaset i as Invoice;

//globālo mainīgo deklarācijas
value storage as Warehouse;
value startDate as DateTime;
value endDate as DateTime;

//likumu deklarēšana (tiek lietoti globālie mainīgie):
atWarehouse(i) :- i.Warehouse == storage;
inPeriod(i) :- i.DealDate>=startDate, i.DealDate<endDate;
buyingAtWarehouseInPeriod(i) :- atWarehouse(i),
    inPeriod(i);

//vaicājuma deklarēšana (tiek lietoti globālie mainīgie):
BuyingAtWarehouseInPeriod(companyWarehouse, startDate,
    endDate) = buyingAtWarehouseInPeriod(i)?
```

Aplūkotajā piemērā tiek izmantoti globālie mainīgie, kas padara kodu sarežģītāk lietojamu paralēlās sistēmās. Turklāt parametru (globālo mainīgo) nosaukumi un tipi tiek atkārtoti pārāk daudz vietās, tādējādi palielinot koda liekvārdību. Koda liekvārdība ir nopietna problēma domēnspecifiskajās valodās, kuras sastāv no koda fragmentiem, kas var tikt interpretēti kā konstrukcijas. Šādi koda fragmenti satur koda daļas, kas var tikt interpretētas kā konstrukciju parametri.

8.5 Vaicājumu valoda, lietojot netiešos parametrus un netiešos argumentus

```
//meta-kopas deklarēšana
metaset i as Invoice;

//likuma deklarēšana, netiešie parametri:
//storage, startDate, endDate:
atWarehouse(i) :- i.Warehouse == storage;
inPeriod(i) :- i.DealDate>=startDate, i.DealDate<endDate;
//netiešie argumenti: storage, startDate, endDate:
buyingAtWarehouseInPeriod(i):-atWarehouse(i), inPeriod(i);
```

```
//Vaicājuma (jautājuma) deklarācija.
//Netiešie argumenti: storage, startDate, endDate
BuyingAtWarehouseInPeriod(Warehouse storage, DateTime startDate,
DateTime endDate) = buyingAtWarehouseInPeriod(i)?
```

Netiešo parametru un netiešo argumentu izmantošana ļauj domāt vairāk likumu kontekstā (deklaratīva pieeja), nevis konstrukciju deklarēšanas vai izpildes kontekstā (procedurāla pieeja). Tai pat laikā netiek zaudētas konstrukciju izmantošanas priekšrocības, jo semantiski likumi var tikt apstrādāti kā konstrukciju deklarācijas.

8.6 Programmēšanas valodu paplašināšana ar abstraktās datu izgūšanas sintaksi

Datu izgūšanas tehnoloģijas spēj nodrošināt vairāk iespēju nekā tikai "SELECT" operatoru. Tās piedāvā arī datu reprezentēšanas operācijas: kārtošanu, grupēšanu, agregāciju un citas operācijas, kas spēj izmainīt izgūstamo datu noformējumu. Eksistējošās datu izgūšanas tehnoloģijas nepiedāvā jaunus veidus kā implementēt "SELECT" un datu reprezentācijas operācijas, bet eksistējošajā pieejā ir problēmas atkārtoti izmantot jau esošos koda fragmentus. Problēmu varētu atrisināt ar abstraktu, deklaratīvu datu izgūšanas sintaksi, kas nesatur datu reprezentācijas operācijas. Šādā datu izgūšanas sintaksē datu reprezentācijas operatori var tikt pielikti klāt kā likumi, padarot datu izgūšanu strukturētāku un palielinot eksistējošo koda daļu atkārtotās izmantošanas iespējas.

9 SECINĀJUMI

1. Promocijas darbā autors ir atklājis jaunas abstrakcijas simbolisko aprēķinu jomā un tās ir: netiešie parametri, netiešie argumenti un Grace~ operators.
2. Netiešie parametri un netiešie argumenti var tikt izmantoti matemātiskajā modelēšanā. Tas rosina domāt, ka netiešo parametru un netiešo argumentu ideja varētu tikt veiksmīgi izmantota izglītības jomā, jo netiešie parametri un netiešie argumenti ļauj padarīt matemātiskās izteiksmes kodolīgākas, tādējādi, lasītājs var koncentrēt vairāk uzmanības kodolīgākai sintaksei ar mazāku liekvārdību un veltīt vairāk laika nozīmīgākajām detaļām.
3. Grace~ operators ir ērts veids kā mainīt netiešo parametru secību, jo tas spēj uzlabot koda lasāmību, paaugstināt koda rediģējamību un samazināt iekļauto lambda izteiksmju liekos blokus.

4. Netiešie parametri, netiešie argumenti un Grace~ operators ir noderīgi programmēšanas valodu izveidei, kurās ir atbalsts apakšprogrammām, kas var pieņemt parametrus. Šādu apakšprogrammu piemēri ir: konstrukcijas (methods), procedūras, konstruktori, lambda izteiksmes un citi elementi. Citiem vārdiem sakot: atklātās metodes kā lietot netiešos parametrus ir noderīgas programmēšanas valodās, kuras atbalsta vismaz vienu konceptu, kas var tikt interpretēts kā konstrukcija. Netiešie parametri un netiešie argumenti paaugstina koda lasāmību; samazina liekvārdību parametru nosaukumu un tipu deklarēšanā, tādējādi padarot atsevišķus koda fragmentus kodolīgākus; ļauj samazināt atkarību no globālajiem mainīgajiem, tādējādi padarot kodu vieglāk pielāgojamu paralēlām skaitļošanas sistēmām.
5. Veicot perfektās lambda sintakses pētījumus, autors identificēja šādas problēmas eksistējošo programmēšanas valodu sintaksēs:

- C# 6.0 lambda sintakse satur vairākas liekas konstrukcijas: tiešā veidā deklarētus parametrus un lieku punktuācijas lietojumu (liekus atdalītājsimbolus).
- Java 8.0 lambda sintakse ir līdzīga C# 6.0 lambda sintaksei, taču papildus tam, java 8.0 atbalsta konceptu sauktu par "pamatkonstrukciju atsaucēm" (method references). "Pamatkonstrukciju atsauces" ļauj dažas lambda izteiksmes padarīt īsākas, taču to trūkums ir simbola "::" izmantošana, jo šī simbola pielietošanai ir šaurs lietojuma gadījumu loks. "Pamatkonstrukciju atsauču" koncepts nav jauna abstrakcija, tas drīzīk ir sintaktisks pārveidojums, kas paaugstina valodas sarežģītību, pretī nesolot ilgtermiņa ieguvumu – spēju novest pie perfektās lambda sintakses.
- Valodas Kotlin operators "it" vienā lambda izteiksmē var tikt izmantots atkārtoti un tas ļauj piekļūt ne tikai pamata konstrukcijām, bet arī konstruktoriem un citiem elementiem. Taču operators "it" spēj darboties tikai viena parametra lambda izteiksmēs (pat "pamatkonstrukciju atsauces" spēj darboties vairāku parametru pamatkonstrukcijās).
- Valoda Clojure ieviesa "parocīgo lambdu" (shorthand lambda), kas padara lambda izteiksmes kodolīgas, bet "parocīgās lambdas" nevar tikt izmantotas iekļautajās anonīmajās funkcijās, lai no iekšējās funkcijas ķermeņa piekļūtu ārējās funkcijas parametriem. Turklāt cilvēka prāts nav piemērots kompleksu algoritmu apstrādei parametru vietā izmantojot indeksus (šāda pieeja samazina koda lasāmību un rediģējamību).

6. Autors konstatēja, ka programmēšanas valodu lambda sintakse var tikt iedalīta šādās četrās kategorijās:

- Lambda sintakse, kas satur tiešā veidā deklarētus parametrus un funkcijas ķermeni (C#, Java).
- Lambda sintakse, kas sastāv no funkcijas ķermeņa, kurā funkcijas parametriem tiek piekļūts izmantojot speciālus atslēgvārdus. Speciālo

atslēgvārdu skaits atbilst atbalstāmo parametru skaitam kodolīgo lambda izteiksmju sintaksē. Piem., Kotlin valodā ir tikai viens šāds atslēgvārds "it" un tas darbojas tikai viena parametra lambda izteiksmēs. Programmēšanas valodā Q ir 3 šādu atslēgvārdu atbalsts: "x" – pirmais parametrs, "y" – otrais parametrs, "z" – trešais parametrs.

- Lambda sintakse, kas sastāv no funkcijas ķermeņa, no kura funkcijas parametriem tiek piekļūts izmantojot speciālu simbolu un indeksu (Clojure, Swift).
- Lambda sintakse, kura sastāv no funkcijas ķermeņa, kurā katrs nezināmais identifikators tiek interpretēts kā funkcijas netiešais parametrs jeb funkcijas parametrs, kas deklarēts netiešā veidā. Netiešo parametru secība atbilst parametru izmantošanas secībai funkcijas ķermenī, bet tā var tikt mainīta, lietojot Grace~ operatoru. (Kat-lang).

7. Netiešie parametri kopā ar Grace~ operatoru var tikt izmantoti, lai uzlabotu lambda izteiksmju sintaksi programmēšanas valodās. Lietojot minētās konstrukcijas, iespējams sasniegt perfekto lambda sintaksi, kas nesatur liekvārdību. Perfektā lambda sintakse padara lambda izteiksmes lasāmākas un uzlabo koda rediģējamības faktoru (Blow J., 2014), ļaujot programmētājiem darboties produktīvāk.
8. Programmēšanas valodās var tikt uzturētas dažādas abstrakcijas un mēdz gadīties, ka valodas sintakse nenodrošina perfektus apstākļus priekš visām uzturētajām abstrakcijām. Var gadīties, ka nesamazinot atbalstu uzturētajām abstrakcijām, programmēšanas valodas sintaksi nevar uzlabot, lai tā spētu saturēt perfekto lambda sintaksi. Līdz ar to vienkāršākais veids, kā implementēt perfekto lambda sintaksi ir radīt jaunu programmēšanas valodu. Tādējādi nav jādomā par savietojamību ar iepriekšējām versijām un citiem ierobežojošiem faktoriem. Tāpēc kā daļa no šī promocijas darba tika radīta Kat-lang jeb abstrakta programmēšanas valoda aprēķinu veikšanai. Kat-lang ir eksperimentāla funkcionālās programmēšanas valoda, kuras sintakse ir līdzīga vidusskolas līmeņa algebras sintaksei. Vienkāršota Kat-lang versija ir implementēta lietotnē "IlCalculus", kas ir pielāgojama, lietotājam draudzīga kalkulatora lietotne. IlCalculus kalkulators atšķiras no konkurējošajām kalkulatora lietotnēm ar savu dizainu. Izmantojot netiešos parametrus, ir iespējams atbrīvoties no "uzlecošajiem" ekrāniem funkciju definēšanā. Netiešo parametru inovācija ļauj lietotājiem izmantot skārienjūtīgā ekrāna iespēju "pavelc un atlaid" (drag & drop) funkciju definēšanā, rediģēšanā un izpildīšanā. Spēja definēt un izpildīt funkcijas vienā ekrānā ir īpaši svarīga lietojot maza ekrāna ierīces.
9. Autors radīja operatoru "memberof", kas spēj iegūt objekta elementa metadatus tipu drošā veidā. Tāpat autors izgudroja kontekst atkarīgu objekta elementu metadatu izgūšanas operatoru "member" un konstrukciju parametru modifikatoru "meta", kas konstrukciju

parametrus kompilatoram liek interpretēt kā elementa metadatu piekļuves izteiksmes nevis kā elementa vērtības piekļuves izteiksmes.

10. Izgudroto metadatu piekļuves operatoru implementēšanai nepieciešamas izmaiņas programmēšanas ietvaros, piem., ietvarā .NET, Java, u.c., kā arī programmēšanas valodu sintaksē.
11. Eksistējošās programmēšanas valodas ļauj kombinējot datu fragmentus, veidojot kompleksas datu struktūras. Dažas programmēšanas valodas ļauj piekļūt metadatiem, kas pēc tam var tikt pilnībā vai daļēji kombinēti kompleksās datu struktūrās, bet neviena no eksistējošajām programmēšanas valodām nepiedāvā valodas līmenī iestrādātu tipu drošu atbalstu metadatu un datu kombinēšanai.
12. Autors definēja: metadatu un datu kombinēšanas rezultāts ir īpašības-ierobežojuma abstrakcija; īpašību-ierobežojumu un metadatu kombinēšanas rezultāts ir meta-kopa jeb datu vaicājuma abstrakcija. Meta-kopas nesatur biznesa objektus un tās nesatur atsauces uz biznesa objektiem; meta-kopas var saturēt tikai ierobežojumus, kuri var tikt izmantoti vaicājumu ģenerēšanā dažādiem datu avotiem un datu izgūšanā.
13. Autors uzlaboja loģiskās programmēšanas dzini, lai tas darbotos nevis ar objektiem (datiem), bet gan ar meta-kopām (metadatiem). Veicot decentralizētu dedukciju ar metadatiem nevis ar datiem, datu avots tiek atdalīts no loģiskās programmēšanas dziņa, tādējādi pārvēršot loģiskās programmēšanas dzini meta-kopu rēķinos jeb otrās kārtas predikātu loģikas paveidā, kur meta-kopas tiek kombinētas kopā, rezultātā iegūstot meta-kopu sarakstu. Meta-kopu saraksts var tikt izmantots automātiskai datu vaicājumu ģenerēšanai. Izstrādātais risinājums ļauj izpildīt dedukciju izkliedētās vidēs, kur dedukcijas process var tikt veikts uz klienta datora, bet datu vaicājumi no klienta datora tiek sūtīti centralizētam datu avotam - serverim.
14. Autora atklātie meta-kopu rēķini var tikt izmantoti, lai aizstātu standarta datu vaicājumu valodu sintaksi ar datu vaicājumu valodas sintaksi, kas līdzīga Prolog valodas sintaksei, tādējādi tiktu uzlabota tipu drošība, palielināta likumu atkārtota izmantošana un samazināta vaicājumu valodas sintakses atkarība no kāda konkrēta datu avota arhitektūras. Šāda pieeja ir ērta vaicājumu veikšanai NoSQL datu avotos, kā arī vaicājumu veikšanai relāciju datu bāzēs.
15. Prolog tipa sintakse, kas spēj darboties ar meta-kopām, var tikt papildināta ar netiešajiem parametriem, netiešajiem argumentiem un Grace~ operatoru.
16. Meta-kopu atbalsts programmēšanas valodu līmenī ir koncepcija, kas atļautu programmēšanas valodās ieviest iespēju definēt tipu drošus vaicājumus.
17. Autors izstrādāja abstraktu datu izgūšanas tehnoloģijas prototipu.

IZMANTOTĀ LITERATŪRA

1. Albahari J., Albahari B. (2012). C# 5.0 in a Nutshell, 5th Edition, The Definitive Reference. O'Reilly Media
2. Atencio L. (2015). Understanding Lambda Expressions. Pieejams: https://medium.com/@luijar/understanding-lambda-expressions-4fb7ed216bc5#.liszqjahd
3. Bancilhon F., Sagiv Y., Ullman J.D., (1986). Magic Sets and Other Strange Ways to Implement Logic Programs. Pieejams: https://web.archive.org/web/20120308104055/http://ssdi.di.fct.unl.pt/krr/docs/magicsets.pdf
4. BashmakovaI.G.; Smirnova G.S.; Shenitzer A. (2000). The Beginnings and Evolution of Algebra. The Mathematical Association of America, USA.
5. Blow. J. (2014). A programming language for games, talk #2
6. Bratko I., (2000). Prolog Programming for Artificial Intelligence. Pearson, UK.
7. C# 6.0 (2015). C# Language Specification, Version 6. Microsoft Corporation. Pieejams: https://github.com/ljw1004/csharpspec/blob/gh-pages/README.md
8. C# Language Specification 5.0 Specification. (2012). Microsoft Corporation. Pieejams: http://www.microsoft.com/en-us/download/details.aspx?id=7029
9. Caller Information (2015). Caller Information (C# and Visual Basic). Microsoft Corporation. Pieejams: http://msdn.microsoft.com/en-us/library/hh534540.aspx
10. Cohen M.S. 2004. Second Order Logic. Pieejams: http://faculty.washington.edu/smcohen/120/SecondOrder.pdf
11. Colburn T., Shute G. (2007). Abstraction in Computer Science. Springer Science+Business Media B.V. 2007. Pieejams: https://www.d.umn.edu/~tcolburn/papers/Abstraction.pdf
12. Demers F.-N., Malenfant J. (1995). Reflection in logic, functional and object-oriented programming: a Short Comparative Study. In: Proc. of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI. Montreal (1995), 29–38.
13. DLR (2009). DRL Expression Tree Specification. Pieejams: http://dlr.codeplex.com/wikipage?title=Docs%20and%20specs&referringTitle=Documentation
14. Elshoff J.L., Marcotty M., (1982). Improving computer program readability to aid modification, Communications of the ACM, v.25 n.8, p.512-521

15. Enderton H.B. (2009). Second-order and Higher-order Logic. Stanford Encyclopedia of Philosophy. Pieejams: https://plato.stanford.edu/entries/logic-higher-order/
16. Erlang (2016). Erlang function declaration syntax. Pieejams: http://www.erlang.org/doc/reference_manual/functions.html
17. Forman I.R., Forman N. (2004). Java Reflection in Action. Manning Publications
18. Gosling J., Joy B., Steele G., Bracha G., Buckley A. (2013). The Java® Language Specification, Java SE 8 Edition; Oracle America, Inc. Pieejams: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf
19. Grace J. (2015). Facebook. Pieejams: https://www.facebook.com/asd930319
20. IlCalculus. 2015. Logics Research Centre. Pieejams: https://www.microsoft.com/en-us/store/p/il-calculus/9wzdncrdmb09
21. Jaffar J., Maher M.J. (2013). Constraint Logic Programming: A Survey. Pieejams: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.4288&rep=rep1&type=pdf
22. Jemerov D, Isakova S. (2017). Kotlin in Action. Manning Publications.
23. Kline, M. (1990). Mathematical Thought from Ancient to Modern Times. New York: Oxford University Press. pp. 35–37. ISBN 0-19-506135-7.
24. Kotlin language reference (2015). JetBrains. Pieejams: https://kotlinlang.org/docs/kotlin-docs.pdf
25. LINQ (2015). LINQ – .NET Language Integrated Query. Microsoft Corporation. Pieejams: http://msdn.microsoft.com/en-us/netframework/aa904594
26. LINQ TO SQL. (2017). Microsoft Corporation. Pieejams: http://msdn.microsoft.com/en-us/library/bb386976.aspx
27. Lippert E. (2009). In Foof We Trust: A Dialogue. Pieejams: http://blogs.msdn.com/b/ericlippert/archive/2009/05/21/in-foof-we-trust-a-dialogue.aspx
28. Martin R.C. (2003). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. p. 95. ISBN 978-0135974445.
29. Mastin L. (2010). 17th Century Mathematics - Leibniz. Pieejams: http://www.storyofmathematics.com/17th_leibniz.html
30. MDA (2013). Model Driven Architecture. Pieejams: http://www.omg.org/mda/
31. Meijer H., Hejlsberg A., Box D., Warren M., Bolognese L., Katzenberger G., Hallam P., and Kulkarni D. (2007). Lambda expressions. US Patent App. 11/193,565.
32. Migliore M. (2011). How to implement MVVM (Model-View-ViewModel) in TDD (Test Driven Development). Pieejams: https://code.msdn.microsoft.com/How-to-implement-MVVM-71a65441

33. Orcas (2006). Microsoft Visual Studio Code Name “Orcas” Language-Integrated Query, May 2006 CTP. Pieejams: http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=11289
34. Palermo J., Bogard J, Hexter E., Hinze M., Skinner J. (2012). ASP.NET MVC 4 in Action. Manning, New York
35. Parkhe R. (2013). Global Variables Are Bad. Pieejams: http://wiki.c2.com/?GlobalVariablesAreBad
36. Paterson J., Edlich S., (2006). The Definitive Guide to db4o. Apress, 2006.
37. Pereira J. (2007). Specifying optional and default values for method parameters. U.S. Patent 2007/0142929 A1, June 21, 2007.
38. Pierce B.C. (2002). Types and programming languages. MIT Press, London, England.
39. Rojas R. (1998), A Tutorial Introduction to the Lambda Calculus. Berlin. Pieejams: http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf
40. Rusina A. (2010). Getting Information About Objects, Types, and Members with Expression Trees. Pieejams: http://blogs.msdn.com/b/csharpfaq/archive/2010/01/06/getting-information-about-objects-types-and-members-with-expression-trees.aspx
41. Scala language documentation. (2017). Pieejams: http://www.scala-lang.org/documentation/
42. Skeet J. C# in Depth, Third Edition, Manning (2013)
43. Smith J. (2009). WPF Apps With The Model-View-ViewModel Design Pattern. Pieejams: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx
44. SODA (2002). Simple Object Database Access. Pieejams: http://sodaquery.sourceforge.net/docs/org/odbms/Query.html
45. Spolsky J. 2002. The Law of Leaky Abstractions. Pieejams: https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/
46. Stansifer R. D. (1994). The study of programming languages. Prentice-Hall, Inc. New Jersey, USA.
47. Swift 3.0 (2017). The swift programming language. Apple Inc. Pieejams: https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html
48. Tashtoush Y. Odat Z., Alsmadi I., and Yatim. M. (2013). Impact of programming features on code readability. International Journal of Software Engineering and Its Applications, 7(6):441 458, 2013.
49. Tayon S. Fingerhut A. (2016). Clojure 1.8 Cheatsheet v35. Pieejams: https://clojure.org/api/cheatsheet

50. Tutorials/Functions (2016). Kx. Pieejams: http://code.kx.com/wiki/Tutorials/Functions
51. USPTO (2017). USPTO Public Pair. The United States Patent and Trademark Office an agency of the Department of Commerce. Pieejams: http://portal.uspto.gov/pair/PublicPair
52. Vanags M. (2015). Grace operator for changing order and scope of implicit parameters, August 16 2016. US Patent 9,417,850.
53. Vanags M., Cevere R. (2017). Type Safe Metadata Combining. Computer and Information Science; Vol. 10, No. 2; 2017. ISSN 1913-8989. Canadian Center of Science and Education. Pieejams: https://doi.org/10.5539/cis.v10n2p97
54. Vanags M., et.al. (2014). Strongly typed metadata access in object oriented programming languages with reflection support. US Patent App. 13/773,662
55. Vanags M., Justs J., Romanovskis D. (2013). Implicit parameters and implicit arguments in programming languages. June 7 2016. US Patent 9,361,071.
56. Vanags M., Licis A., Justs J. (2013). Strongly typed metadata access in object oriented programming languages with reflection support. Baltic J. Modern Computing, Vol. 1 (2013), No. 1, 77-100
57. Vanags M., Licis A., Justs J. (2014). Abstract, structured data store querying technology. US Patent App. 13/781,804
58. Veitch J. (1998). "A history and description of CLOS". In Salus, Peter H. Handbook of programming languages. Volume IV, Functional and logic programming languages (first ed.). Indianapolis, IN: Macmillan Technical Publishing. pp. 107–158. ISBN 1-57870-011-6.
59. Wesdyer (2007). Anonymous Recursion in C#. Pieejams: http://blogs.msdn.com/b/wesdyer/archive/2007/02/02/anonymous-recursion-in-c.aspx
60. Душкин Р.В. (2006). Функциональное программирование на языке Haskell. ДМК Пресс.