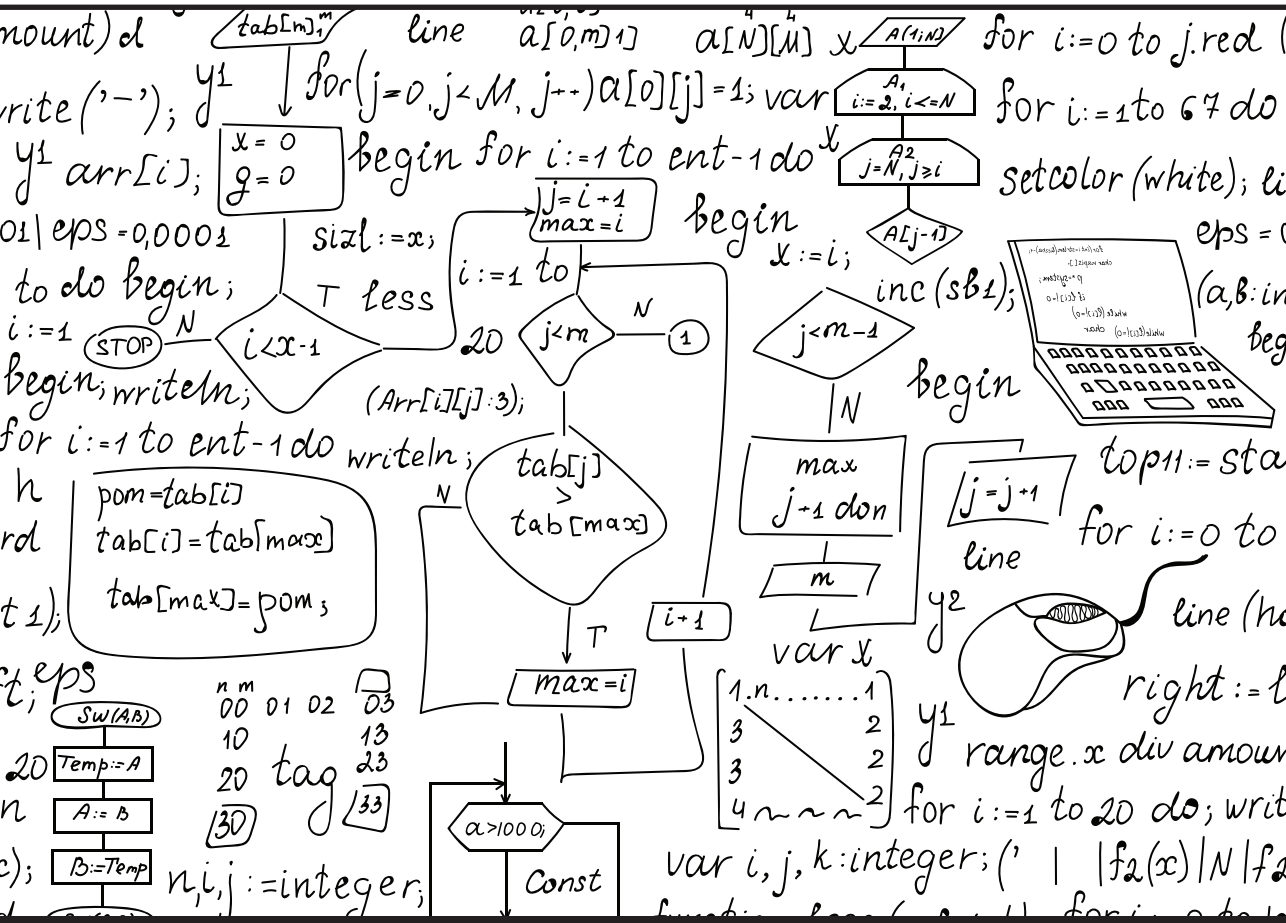
RĪGAS TEHNISKĀ UNIVERSITĀTE

Ruslan Batdalov

OBJEKTORIENTĒTO TIPU SISTĒMAS IZSTRĀDE, LIETOJOT PROJEKTĒŠANAS ŠABLONU METODOLOĢIJU

Promocijas darba kopsavilkums



RTU Izdevniecība
Rīga 2024

RĪGAS TEHNISKĀ UNIVERSITĀTE
Datorzinātnes, informācijas tehnoloģijas un enerģētikas fakultāte

Ruslan Batdalov
Doktora studiju programmas "Datorzinātne un informācijas tehnoloģija" doktorants

OBJEKTORIENTĒTO TIPU SISTĒMAS IZSTRĀDE, LIETOJOT PROJEKTĒŠANAS ŠABLONU METODOLOĢIJU

Promocijas darba kopsavilkums

Zinātniskā vadītāja
profesore *Dr. sc. ing.*
OKSANA ŅIKIFOROVA

RTU Izdevniecība
Rīga 2024

Batdalov R. Objektorientēto tipu sistēmas izstrāde, lietojot projektēšanas šablonu metodoloģiju. Promocijas darba kopsavilkums. – Rīga: RTU Izdevniecība, 2024. – 44 lpp.

Publicēts saskaņā ar promocijas padomes "RTU P-07" 2024. gada 24. maija lēmumu, protokols Nr. 1.

Vāka attēls no *www.shutterstock.com*

https://doi.org/10.7250/9789934371103
ISBN 978-9934-37-110-3 (pdf)

PROMOCIJAS DARBS IZVIRZĪTS ZINĀTNES DOKTORA GRĀDA IEGŪŠANAI RĪGAS TEHNISKAJĀ UNIVERSITĀTĒ

Promocijas darbs zinātnes doktora (*Ph. D.*) grāda iegūšanai tiek publiski aizstāvēts 2024. gada 7. oktobrī Rīgas Tehniskās universitātes Datorzinātnes, informācijas tehnoloģijas un enerģētikas fakultātē, Zunda krastmalā 10, 104. auditorijā.

OFICIĀLIE RECENZENTI

Profesors *Dr. habil. sc. ing.* Jānis Grundspeņķis,
Rīgas Tehniskā universitāte

Profesore *Dr. sc. ing.* Irina Arhipova,
Latvijas Biozinātņu un tehnoloģiju universitāte, Latvija

Profesors *Dr.* Horhe Luiss Ortega Arhona (*Jorge Luis Ortega Arjona*),
Meksikas Nacionālā autonomā universitāte, Meksika

APSTIPRINĀJUMS

Apstiprinu, ka esmu izstrādājis šo promocijas darbu, kas iesniegts izskatīšanai Rīgas Tehniskajā universitātē zinātnes doktora (*Ph. D.*) grāda iegūšanai. Promocijas darbs zinātniskā grāda iegūšanai nav iesniegts nevienā citā universitātē.

Ruslan Batdalov ……. (paraksts)
Datums: ……

Promocijas darbs ir uzrakstīts angļu valodā, tajā ir ievads, sešas nodaļas, secinājumi, literatūras saraksts, 23 attēli, trīs tabulas, divi pielikumi, kopā 148 lappuses, neieskaitot pielikumus. Literatūras sarakstā ir 128 nosaukumi.

SATURS

IEVADS

Projektēšanas šabloni (angl. *design patterns*) tiek plaši izmantoti programmatūras inženierijā un izstrādē. Šabloni nodrošina atkārtojošos problēmu standarta risinājumus, dokumentē pārbaudītos programmatūras projektējumus un arhitektūru, nosaka augstāka līmeņa abstrakcijas un atvieglo projektējumu paziņošanu, dodot kopīgu vārdu krājumu (*Buschmann et al.*, 2013). Franks Bušmans u. c. norādīja, ka jebkurš netriviāls projektējums neizbēgami ietver daudzus šablonus, apzināti vai nē (*Buschmann et al.*, 2007b).

Plaši atzīts šablonos balstītu projektējumu trūkums ir to sarežģītība. Lai gan problēmas risināšana neizbēgami rada zināmu sarežģītību, projektēšanas šablonu nekritiska piemērošana bieži rada risinājumus, kas ir sarežģītāki, nekā nepieciešams. Tas prasa atrast netriviālu kompromisu starp lietošanai gatava risinājuma izmantošanas vienkāršību un iegūtā projektējuma sarežģītību.

Ērihs Gamma *et al.* brīdināja par šīm briesmām vēl projektēšanas šablonu ēras sākumā (*Gamma et al.*, 1995), un notikumu gaita liecināja, ka šīs bažas bija pamatotas. Franks Bušmans u. c. sērijas "Šablonorientētā programmatūras arhitektūra" pirmajā grāmatā apgalvoja, ka šabloni palīdz izstrādātājiem pārvaldīt programmatūras sarežģītību (*Buschmann et al.*, 2013), bet piektajā grāmatā atzina, ka daudzas projektēšanas kļūmes izraisīja nevajadzīga un neattaisnota arhitektūras sarežģītība, neskatoties uz apzināto un skaidro projektēšanas šablonu lietošanu (vai pat tās dēļ) (*Buschmann et al.*, 2007b). Pēters Sommerlads, tās pašas sērijas līdzautors, gāja vēl tālāk un atklāti apgalvoja, ka projektēšānas šabloni ir slikti programmatūras projektēšanai šīs pārmērīgās sarežģītības dēļ (*Sommerlad*, 2007).

Promocijas darba autors izvirzīja hipotēzi, ka šablonos balstītu risinājumu pārmērīgo sarežģītību daļēji izraisa esošo programmēšanas valodu nepietiekams izteiksmīgums (*Batdalov*, 2016). Pastāv starpība starp šabloniem, kas atspoguļo domu konstrukcijas, ar kurām programmētāji spriež par savām programmām, un programmēšanas valodas līdzekļiem, kas vēsturiski ir attīstījušās no to pamatā esošajiem tehniskajiem līdzekļiem. Rezultātā programmēšanas valodu nepietiekamais izteiksmīgums (kas mēra ideju plašumu, ko programmētāji var izteikt, lietojot valodu (*Leitão and Proença*, 2014)), padara šablonos balstītu projektējumu realizāciju par sarežģītāku, nekā nepieciešams.

Realizācijas grūtības ir tikai viens no daudzajiem sarežģītības avotiem. Citi avoti aptver, piemēram, lieko elastību, nepiemērotu abstrakciju izmantošana vai problēmai raksturīgā sarežģītība. Taču valodas izteiksmīguma problēma atšķirībā no citām minētajām nav specifiska konkrētai problēmai un tās risinājumam. Tāpēc šīs problēmas risināšana var palīdzēt vairākās situācijās.

Promocijas darbā tiek pētīts, kā var uzlabot programmēšanas valodu izteiksmīgumu, lai to tuvinātu projektēšanas šablonos attēlotām domu konstrukcijām. Programmēšanas valodas izteiksmīguma paaugstināšana ir nebeidzams process, jo cilvēki vienmēr var izgudrot jaunas augstāka un augstāka līmeņa abstrakcijas. Tomēr piedāvātā pieeja izmanto projektēšanas šablonus kā virzienu, kurā programmēšanas valodas varētu attīstīties. Tas vienkāršotu šablonos balstītu projektējumu realizāciju.

Būtisks projektējuma sarežģītības aspekts ir esošās sistēmas attīstība. Šablona ieviešana esošā sistēmā ir sarežģīta, un tā novēršana var būt vēl grūtāka (*Sommerlad*, 2007). Lai risinātu šo problēmu, promocijas darbā tiek mēģināts pēc iespējas vairāk vispārināt pētītās konstrukcijas. Tas atvieglo sistēmas attīstību, jo vispārējā gadījuma apakštipa vai realizācijas aizstāšana ar citu parasti ir vienkāršāka nekā pāreja uz nesaistītu tipu.

Lai panāktu nepieciešamo vispārinājumu, pašas apskatītās konstrukcijas tiek aprakstītas kā šabloni. Dažādās programmēšanas valodās izmantoto konstrukciju salīdzināšana un visu šablona komponentu noteikšana ļauj atrast to vispārīgo gadījumu. Tādējādi šabloni kalpo kā primārais promocijas darba metodiskais līdzeklis.

Reālā programmēšanas valoda, kuras pamatā būtu noteiktās vispārinātās konstrukcijas, ir ārpus promocijas darba tvēruma. Tās vēlamais rezultāts ir teorētisks pamats šādai valodai, formālo tipu sistēma. Tipu sistēmā tiek izmantoti tipu teorijas F_ω rēķinu formālismi (*Pierce*, 2002), kas nodrošina promocijas darba mērķu sasniegšanai pietiekamu universālo un eksistenciālo tipu atbalstu. Šo tipu mērķis ir būt formāliem vispārinātu konstrukciju, ko izmanto spriešanās par programmām, attēlojumiem.

PRIEKŠMETA AKTUALITĀTE

Jaunas programmēšanas valodas parādās lielā tempā. Dažas salīdzinoši nesen radītas valodas, kurās tiek izmantotas jaunas pieejas un kas jau ir guvušas lielu popularitāti, ir *Scala*, *Kotlin*, *Go*, *TypeScript* un *Rust*. Pat *Java* valodas veidotāji atzīst plaši izplatīto vēlmi iegūt "nākamo dižo valodu" (*Gosling et al.*, 2015). Turklāt daudzas pētniecības valodas ir izveidotas, lai pārbaudītu jaunas pieejas, koncepcijas un iespējas pirms to ieviešanas galvenajās valodās. Šī situācija parāda, ka konceptuālu uzlabojumu meklēšana programmēšanas valodu jomā notiek pastāvīgi.

Arī vecākas programmēšanas valodas nepaliek nemainīgas. Tādas valodas kā *C++* un *Java*, kā arī citas, piedzīvo būtiskas izmaiņas starp versijām, aizņemoties koncepcijas no citām valodām un ieviešot jaunas. Tiekšanās pēc augstāka izteiksmīguma ir būtisks pārmaiņu virzītājspēks gan jaunās, gan esošās programmēšanas valodās (*Batdalov*, 2017).

Projektējumu sarežģītības temats šablonu kopienā pēdējā laikā nav tik daudz apspriests kā iepriekš. Tomēr problēma drīzāk tika pieņemta, nevis atrisināta. Šablonu kopiena galvenokārt koncentrējas uz jaunu šablonu atklāšanu, turpretim jomas konceptuāla pārskatīšana nav notikusi jau ilgu laiku. Iepriekš atzītā šablonos balstītu projektējumu pārmērīgas sarežģītības problēma joprojām pastāv.

Promocijas darbs atbalsta programmēšanas valodu virzību uz augstāku izteiksmīgumu un daļēji risina šablonos balstītu projektējumu sarežģītības problēmu. Tas arī daļēji sistematizē notiekošos procesus programmēšanas valodu attīstībā, jo darbā aprakstītie tipi bieži vispārina nesenus jaunumus. Tādējādi promocijas darbs atbilst programmēšanas valodu attīstības un projektēšanas šablonu pašreizējām vajadzībām.

PROMOCIJAS DARBA MĒRĶIS

Promocijas darba mērķis ir izstrādāt tipu sistēmu, kas vispārina esošas programmēšanas valodu konstrukcijas līdz spriešanai par programmām tipisko domu konstrukciju abstrakcijas līmenim, tādējādi paaugstinot to izteiksmīgumu, salīdzinot ar esošajām valodām.

PROMOCIJAS DARBA UZDEVUMI

Lai sasniegtu promocijas darba mērķi, tika noteikti vairāki uzdevumi. Tie ir:

1. Izanalizēt programmēšanas valodu evolūcijas tendences un iepriekš aprakstītās grūtības projektēšanas šablonu realizācijā.
2. Formulēt prasības tipu sistēmai, pamatojoties uz minētajām tendencēm un grūtībām.
3. Aprakstīt izplatītākos datu salikšanas šablonus un skaitļošanas pamatprimitīvus.
4. Formalizēt aprakstītos šablonus attēlojošas tipu teorijas konstrukcijas.
5. Izstrādāt vajadzības pēc izteiksmīgākām konstrukcijām ilustrējošu piemēra projektu.
6. Pārbaudīt, ka piedāvāto tipu augstāks izteiksmīgums vienkāršotu piemēra projekta realizāciju un attīstību.

PĒTĪJUMA OBJEKTS

Promocijas darba pētījuma objekts ir programmēšanas valodu līmeņa konstrukcijas attēlojošie abstrakti jēdzieni.

PĒTĪJUMA PRIEKŠMETS

Promocijas darba pētījuma priekšmets ir izplatīti programmēšanas valodu līmeņa pamatkonstrukcijas vispārinoši šabloni un to tipu teorijas formalizācija.

PĒTĪJUMU METODES

Promocijas darbā tika izmantotas trīs metodes.

1. Programmēšanas valodu salīdzinošā analīze, lai noteiktu līdzīgas valodas līmeņa konstrukcijas un aprakstītu to vispārīgās formas.
2. Noteikto konstrukciju tipu teorijas formalizācija, lietojot F_ω rēķinu formālismus.
3. Domu eksperiments par izstrādāto teorētisko konstrukciju lietojamību praktiskā projektā.

ZINĀTNISKAIS JAUNIEGUVUMS

1. Tipiskas programmēšanas valodas līmeņa konstrukcijas attēlojoši jēdzieni ir aprakstīti kā projektēšanas šabloni.
2. Projektēšanas šablonu valodas un struktūras lietošana ļāva vispārināt minētās konstrukcijas.
3. Noteiktie šabloni ir formalizēti, lietojot tipu teorijas aparātu.

PĒTĪJUMA PRAKTISKĀ NOZĪME

Pārbaudes rezultāti parāda, ka pētījumā piedāvātās valodas līmeņa iespējas vienkāršo praktisko programmatūras izstrādi un attīstību noteiktās situācijās. Šī pētījuma gaitā izstrādāto metodoloģiju nākotnē var piemērot citām valodu un bibliotēku iespējām.

PĒTĪJUMA REZULTĀTU APROBĀCIJA

Promocijas darba rezultāti ir atspoguļoti astoņās publikācijās starptautiskos un Latvijas Zinātnes padomes atzītos žurnālos un izdevumos.

1. Ruslan Batdalov. Inheritance and class structure. In Pavel P. Oleynik, editor, *Proceedings of the First International Scientific-Practical Conference Object Systems – 2010*, pages 92–95, 2010. URL https://cyberleninka.ru/article/n/inheritance-and-class-structure/pdf.
2. Ruslan Batdalov. Is there a need for a programming language adapted for implementation of design patterns? *In Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP '16)*, pages 34:1–34:3. Association for Computing Machinery, 2016. ISBN 978-1-4503-4074-8. doi: 10.1145/3011784.3011822.
    - Indeksēts *Scopus*.
3. Ruslan Batdalov and Oksana Nikiforova. Towards easier implementation of design patterns. In *Proceedings of the Eleventh International Conference on Software Engineering Advances (ICSEA 2016)*, pages 123–128. IARIA, 2016
    - Autora personīgais ieguldījums: saistīto darbu analīze, pieejas izstrāde un piedāvāto iespēju apraksts.
4. Ruslan Batdalov, Oksana Nikiforova, and Adrian Giurca. Extensible model for comparison of expressiveness of object-oriented programming languages. *Applied Computer Systems*, 20(1):27–35, 2016. doi: 10.1515/acss-2016-0012.
    - Autora personīgais ieguldījums: saistīto darbu analīze, salīdzināšanas modeļa izstrāde un salīdzināšanas veikšana.
    - Indeksēts *Web of Science*.
5. Ruslan Batdalov and Oksana Ņikiforova. Implementation of a MIX emulator: A case study of the Scala programming language facilities. *Applied Computer Systems*, 22(1):47–53, 2017. doi: 10.1515/acss-2017-0017.
    - Autora personīgais ieguldījums: saistīto darbu analīze, emulatora projektējums un realizācija, kā arī Scala valodas iespēju analīze.
    - Indeksēts *Web of Science*.
6. Ruslan Batdalov and Oksana Nikiforova. Three patterns of data type composition in programming languages. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18)*, pages 32:1–32:8. Association for Computing Machinery, 2018. ISBN 978-1-4503-6387-7. doi: 10.1145/3282308.3282341.
    - Autora personīgais ieguldījums: saistīto darbu analīze un piedāvāto šablonu apraksts.
    - Indeksēts *Scopus*, *Web of Science*.

7. Ruslan Batdalov and Oksana Nikiforova. Elementary structural data composition patterns. In *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP '19)*, pages 26:1–26:13. Association for Computing Machinery, 2019. ISBN 978-1-4503-6206-1. doi: 10.1145/3361149.3361175.
    - Autora personīgais ieguldījums: saistīto darbu analīze un piedāvāto šablonu apraksts.
    - Indeksēts *Scopus*, *Web of Science*.
8. Ruslan Batdalov and Oksana Nikiforova. Patterns for assignment and passing objects between contexts in programming languages. In *Proceedings of the 26th European Conference on Pattern Languages of Programs (EuroPLoP '21)*, pages 4:1–4:9. Association for Computing Machinery, 2021. ISBN 978-1-4503-8997-6. Doi: 10.1145/3489449.3489975.
    - Autora personīgais ieguldījums: saistīto darbu analīze un piedāvāto šablonu apraksts.
    - Indeksēts *Scopus*, *Web of Science*.

Promocijas darba galvenie rezultāti prezentēti astoņās starptautiskās zinātniskās konferencēs.

1. Pirmā starptautiskā zinātniskā un praktiskā konference Objektsistēmas – 2010, 10.–12.05.2010. – Rostova pie Donas, Krievija, "Inheritance and class structure".
2. EuroPLoP '16 – 21. Eiropas konference par programu šablonu valodām, 06.–10.07.2016. – Irsee, Vācija, "Is there a need for a programming language adapted for implementation of design patterns?".
3. ICSEA 2016 – 11. starptautiskā konference par programmatūras inženierijas attīstību, 21.–25.08.2016. – Roma, Itālija, "Towards easier implementation of design patterns".
4. Rīgas Tehniskās universitātes 57. starptautiskā zinātniskā konference, 13.–16.10.2016. – Rīga, Latvija, "Extensible model for comparison of expressiveness of object-oriented programming languages".
5. Rīgas Tehniskās universitātes 58. starptautiskā zinātniskā konference, 12.–15.10.2017. – Rīga, Latvija, "Implementation of a MIX emulator: A case study of the Scala programming language facilities".
6. EuroPLoP '18 – 23. Eiropas konference par programu šablonu valodām, 04.–08.07.2018. – Irsee, Vācija, "Three patterns of data type composition in programming languages".
7. EuroPLoP '19 – 24. Eiropas konference par programu šablonu valodām, 03.–07.07.2019. – Irsee, Vācija, "Elementary structural data composition patterns".
8. EuroPLoP '21 – 26. Eiropas konference par programu šablonu valodām, 07.–11.07.2021. – Grāca, Austrija, "Patterns for assignment and passing objects between contexts in programming languages".

AIZSTĀVĒŠANAI IR IZVIRZĪTAS DIVAS TĒZES. TĀS IR:

1. Projektēšanas šablonu metodoloģija ir piemērojama valodas līmeņa konstrukciju vispārināšanai šablonu veidā.

2. Aprakstītie šabloni ir formalizējami kā tipi (kas potenciāli ļauj tos pārvērst programmēšanas valodu konstrukcijās) un var atvieglot programmatūras sistēmas izstrādi.

DARBA STRUKTŪRA

1. nodaļā aprakstīts šablonu un tipu teorijas darba konteksts, kā arī izklāstīta pētījuma metodoloģija. 2. nodaļā aprakstītas programmēšanas valodu attīstības tendences un noteiktas saistītās prasības izstrādātajai tipu sistēmai. 3. nodaļā aprakstīti datu salikšanas šabloni un to tipu teorijas formalizācija. 4. nodaļā aprakstīti skaitļošanas šabloni, izņemot piešķiri, un to tipu teorijas formalizācija. 5. nodaļā aprakstīti piešķires šabloni un to tipu teorijas formalizācija. 6. nodaļa parāda izstrādātās tipu sistēmas lietojamību programmatūras izstrādes projektā. Noslēgumā ir apkopots darbs un aprakstīti turpmākie pētījumu virzieni. 1. pielikumā dotas promocijas darbā lietoto terminu definīcijas. 2. pielikumā dots aprakstīto šablonu saraksts ar to lietošanas piemēriem dažādās programmēšanas valodās.

1. IEPRIEKŠĒJIE PĒTĪJUMI UN MOTIVĀCIJA

Promocijas darbs pēta saistību starp projektēšanas šabloniem un programmēšanas valodu elementārajām konstrukcijām. Šajā kontekstā projektēšanas šabloni ir svarīgi, lai atklātu konkrētas problēmas vispārīgu gadījumu un risinājumu. Autors savā maģistra darbā apgalvoja, ka esošās programmēšanas valodas bieži atbalsta noteikto šablonu tikai atsevišķus gadījumus, kas rada grūtības, ja ir nepieciešams nedaudz atšķirīgs risinājums (*Batdalov*, 2017). Šajā situācijā vispārīgā gadījuma kā šablona aprakstīšana var palīdzēt izprast programmēšanas valodas vēlamās iespējas.

Tomēr projektēšanas šabloniem trūkst programmēšanas valodu iespējām nepieciešamās formalitātes. Tipu teorija nodrošina rīku kopu, lai formāli definētu valodas iespējas (*Pierce*, 2002). Promocijas darbā šablonu veidā aprakstītie vispārinājumi pēc tam tiek formalizēti tipu teorijas valodā, lai panāktu nepieciešamo stingrību.

Šajā nodaļā aprakstīta projektēšanas šablonu pieeja, vispārīgie tipu teorijas jēdzieni, saistītie pētījumi un tas, kā šīs pieejas tiek lietotas kopā, lai sasniegtu promocijas darba mērķus.

1.1. Projektēšanas šabloni

Projektēšanas šablonu galvenais uzdevums ir nodrošināt standarta risinājumus bieži sastopamām problēmām programmatūras projektēšanā. Daudzi izplatīti šabloni ir aprakstīti literatūrā lietošanai gandrīz gatavā formā, piemēram, fasāde (angl. *Façade*), rūpnīcas metode (angl. *Factory Method*), iterators (angl. *Iterator*), apmeklētājs (angl. *Visitor*) (*Gamma et al.*, 1995), brokeris (angl. *Broker*), izdevējs-abonents (angl. *Publisher-Subscriber*) (*Buschmann et al.*, 2013).) un citi. Papildus programmatūras projektējumu komponentu lomai tie nodrošina kopīgu valodu, lai izstrādātājiem paziņotu projektēšanas lēmumus. Papildus klasiskajām populārajām grāmatām literatūrā ir aprakstīti tūkstošiem citu šablonu (*Booch*, 2007). Tie ir mazāk zināmi, bet tomēr noderīgi kā projektējumu celtniecības bloki.

Projektēšanas šablonu pieeja nāk no ēku arhitektūras. 1977. gadā Kristofers Aleksanders u. c. piedāvāja lietot šablonus kā arhitektu kopīgu valodu, paužot plaši izmantotus arhitektūras risinājumus (*Alexander et al.*, 1977). Pēc viņu domām, "katrs šablons apraksta problēmu, kas mūsu vidē atkārtojas atkal un atkal, un pēc tam apraksta šīs problēmas risinājuma būtību tādā veidā, ka jūs varat izmantot šo risinājumu miljons reižu, nekad nedarot to vienādi divreiz" (*Alexander et al.*, 1977). Tāda pati pieeja ir piemērojama programmatūras projektēšanas šablonu jomā. Programmatūras projektēšanas šablonus popularizēja Ēriha Gammas u. c. fundamentāla grāmata "Projektēšanas šabloni" (*Gamma et al.*, 1995). Citi ievērojami projektēšanas šabloniem veltīti darbi ir grāmatu sērija "Šablonorientētā programmatūras arhitektūra" (*Buschmann et al.*, 2007a,b, 2013; *Kircher and Jain*, 2013; *Schmidt et al.*, 2013), Džeimsa O. Kopliena "Programmatūras šabloni" (*Coplien*, 1996), Martina Foulera "Uzņēmuma lietojumprogrammu arhitektūras šabloni" (*Fowler*, 2012) un citi. Ikgadējo konferenču par programmu šablonu valodām (*PLoP*, *EuroPLoP*, *AsianPLoP* un citas) materiāli ietver daudzu citu projektēšanas šablonu un šablonu valodu (savstarpēji saistītu šablonu kopu,

ko izmanto kopā vienā jomā) aprakstus. Visi šie avoti apraksta daudzus šablonus, kas atklāti dažādās programmatūras sistēmās un gatavi atkārtotai izmantošanai.

Projektēšanas šablonus definīcijas atšķiras, taču parasti tās sniedz tās pašas idejas. Ērihs Gamma *et al.* norādīja, ka šabloni ir "sazinājošos objektu un klašu apraksti, kas ir pielāgoti vispārīgas projektēšanas problēmas risināšanai konkrētā kontekstā" (*Gamma et al.*, 1995). Franks Bušmans u. c. piedāvāja līdzīgu definīciju: "Programmatūras arhitektūras šablons apraksta konkrētu atkārtojošos projektēšanas problēmu, kas rodas konkrētos projektējumu kontekstos un piedāvā labi pārbaudītu vispārīgu shēmu tās risināšanai. Risinājuma shēma ir definēta, aprakstot tās sastāvdaļas, to pienākumus un attiecības, kā arī veidus, kādos tie sadarbojas" (*Buschmann et al.*, 2013). Abas definīcijas uzsver tādus projektēšanas šablona galvenos atribūtus kā konteksts, problēma un problēmas risinājums.

Šablonu lietošanas primārais mērķis, ko Ērihs Gamma *et al.* ievietoja savas grāmatas apakšvirsrakstā, ir atkārtota izmantošana (*Gamma et al.*, 1995). Franks Bušmans u. c. noteica šādas šablonu lomas programmatūras arhitektūrā: esošās labākās prakses dokumentēšana; abstrakciju noteikšana un definēšana; kopīgs projektēšanas jēdzienu vārdu krājums un izpratne; programmatūras arhitektūras dokumentēšana; programmatūras ar labi definētām īpašībām izveides atbalsts un pieredzes iegūšana forma, kas var būt neatkarīga no konkrētām projekta detaļām un ierobežojumiem, realizācijas paradigmas un bieži pat no programmēšanas valodas (*Buschmann et al.*, 2007b).

Svarīgs aspekts ir tas, ka šabloni pastāv sistēmās pat tad, kad tas nav apzināts lēmums. Greidijs Bučs norādīja, ka katrai izstrādes kultūrai ir tendence laika gaitā konverģēt uz arhitektūras šablonu kopumu (*Booch*, 2007). Pēc Franka Bušmana u. c. domām, jebkurā netriviālā projektējumā tiek izmantoti daudzi modeļi, **apzināti vai nē** (*Buschmann et al.*, 2007b). Tāpat šablonu pētnieki parasti runā par šablonu atklāšanu, nevis izgudrošanu (*Buschmann et al.*, 2007b, 2013). Šablons ir jāizmanto reālās sistēmās un jāpārbauda pirms aprakstīšanas. Tas padara aprakstīto risinājumu par šablonu, nevis *ad hoc* risinājumu.

Šablonu lietošanai ir arī sava cena. Franks Bušmans u. c. minēja šādus izplatītos slazdus un lamatas: kārdinājums visu pārvērst šablonos; atkārtoti nelietojamu vai nepārbaudītu projektējumu aprakstīšana kā šablonu; šablona uzskatīšana par fiksētu un nemaināmu risinājumu, vadlīniju, kas neietver problēmas risinājumu, aprakstīšana; nepareizs šablona lietojums; pārliecība, ka šablonu mehāniska lietošana nodrošina labu arhitektūru visos gadījumos; radošuma trūkums; pārāk lielas cerības; šablonu lietošanas pilnīgas automatizācijas neiespējamība; nekomponentējamība un šablonu izmantošana pārstrukturēšanas vietā vai otrādi (*Buschmann et al.*, 2007b). Šīs problēmas piespieda Franku Bušmanu u. c. sērijas "Šablonorientētā programmatūras arhitektūra" piektajā sējumā atzīt, ka daudzas sistēmas, kurās šabloni tika izmantoti apzināti un skaidri, beidzās ar nevajadzīgi sarežģītu arhitektūru (*Buschmann et al.*, 2007b). Tomēr, neskatoties uz šīm problēmām, šablonu lietošana sniedz ievērojamas priekšrocības un būtiski atvieglo programmatūras projektēšanu.

1.2. Tipu teorija

Tipu teorija kā datorzinātnes nozare pēta programmēšanas valodās esošās tipu sistēmas. Programmēšanas valodās tipi (piemēram, veseli skaitļi, virknes, masīvi, kartes un bibliotēkas un lietotāja definētas klases) tiek izmantoti, lai definētu atļautās darbības ar noteiktām vērtībām. Tas aizsargā programmas pret ar datu neatbilstošu izmantošanu saistītām kļūdām kompilēšanas vai izpildlaikā.

Tipu teorija nodrošina formālu pamatojumu spriešanai par šiem programmēšanas valodu veiktajiem aizsardzības pasākumiem. Tā veido pamatu rēķiniem (angl. *calculi*), ko izmanto, lai pierādītu formālus apgalvojumus par programmēšanas valodām. Tādā veidā var formāli pierādīt, ka programmēšanas valodas noteikumi noteiktās situācijās ir pietiekami vai nē.

Bendžamins Pīrss tipu sistēmu definēja kā izsekojamu sintaktisku metodi, lai pierādītu noteiktu programmas darbību neesamību, klasificējot frāzes atbilstoši to aprēķināto vērtību tipiem (*Pierce*, 2002). Šī definīcija aptver dažas būtiskas tipu sistēmas īpašības: lietojums programmām; balstīšanās valodas frāžu klasificēšanā; konservativitāte (spēja pierādīt sliktas darbības trūkumu, bet ne tās esamību); ierobežotība ar noteiktiem kļūdu veidiem un izsekojamība (iespēja automātiski pārbaudīt tipu noteikumus) (*Pierce*, 2002). Tipu sistēmu lietošana sniedz šādas priekšrocības: kļūdu agrīna atklāšana; palīdzība uzturēšanā un pārstrukturēšanā; abstrahēšana; dokumentēšana; valodas drošums; efektivitāte un drošība (*Pierce*, 2002).

Tipu var piešķirt simbolam programmā (kompilēšanas laika tips, angl. *compile-time type*) vai vērtībai, ko šis simbols satur (izpildlaika tips, angl. *run-time type*). Valodas atšķiras pēc šo tipu lietojuma. Piemēram, statiski tipētās valodas būtiski paļaujas uz kompilēšanas laika tipiem, kompilēšanas laikā veic pēc iespējas vairāk pārbaužu un bieži nesaglabā tipu informāciju izpildlaikā (lai gan tādas iespējas kā apakštipa polimorfisms noteiktos gadījumos to aizliedz). Turpretim dinamiski tipētās valodas parasti neatbalsta kompilēšanas laika tipus un visas pārbaudes veic izpildlaikā, pamatojoties uz izpildlaika tipu informāciju. Nedaudz paradoksāli, ka visas dinamiski tipētās valodas ir drošas savu abstrakciju aizsargāšanā, bet tas ne vienmēr attiecas uz statiski tipētām valodām (*Pierce*, 2002). Tajā pašā laikā statiski tipētās valodas nodrošina kompilēšanas laika pārbaužu efektivitāti. Tādējādi gan kompilēšanas laika, gan izpildlaika pārbaudēm ir priekšrocības, un tās ir vērtīgas dažādās situācijās.

Pētnieki parasti izšķir pamata (vai neinterpretētos) tipus (piemēram, Būla vērtības, skaitļus un rakstzīmes) un saliktos tipus (piemēram, masīvus, kartes un ierakstus) (*Pierce*, 2002). Pamata tipi ir definēti valodas specifikācijā iepriekš, savukārt saliktajiem tipiem ir noteikti tikai konstruēšanas veidi. Tomēr valoda var zināt par dažiem standarta bibliotēkā definētiem saliktiem tipiem (piemēram, iteratoriem) un atbalstīt šos tipus izmantojošas sintaktiskās konstrukcijas. Robeža starp pamata un saliktajiem tipiem nav universāla, un pamata tips vienā valodā var būt salikts tips citā (piemēram, virkne).

Attiecībā uz saliktajiem datu tipiem tipu teorija atšķir nominālās un strukturālās tipu sistēmas. Tradicionāli tipu teorijā galvenokārt tiek aplūkotas strukturālās tipu sistēmas, kurās tipa nosaukums ir tikai saīsinājums, un strukturāli vienādi tipi ir tas pats tips (*AbdelGawad*, 2017). Tomēr lielākā daļa programmēšanas valodu izmanto nominālās tipu sistēmas, kurās

strukturāli vienādi tipi ar dažādiem nosaukumiem ir atšķirīgi (*AbdelGawad*, 2017). Nominālo tipu sistēmu priekšrocības ietver izpildlaika tipa informācijas pieejamību, dabisku rekursīvo tipu atbalstu, vienkāršu apakštipu pārbaudi un strukturāli saderīgu, bet semantiski atšķirīgu tipu atšķiršanu (*Pierce*, 2002). Tomēr dažas uzlabotas iespējas (piemēram, ģeneriskie tipi) ir grūti ieviest nominālajās tipu sistēmās, tāpēc programmēšanas valodas izmanto nominālo un strukturālo tipu sistēmu hibrīdus (*Pierce*, 2002).

Konkrēti tipi un to semantika nosaka rēķinus, dodot iespēju spriest par programmām un tipu sistēmām. Ir dažādi rēķini dažādām programmēšanas paradigmām un citām valodas īpašībām. Vispārējus rēķinus var arī pielāgot konkrētai valodai. Rēķini ļauj pierādīt tipu sistēmas īpašības, taču tā nav to vienīgā loma. *Scala* valodā vismaz divas iespējas (klases eksplicītais sevis tips (*Odersky et al.*, 2006) un literārie tipi (*Leontiev et al.*, 2019)) vispirms parādījās formālajā valodas semantikas aprakstā un tikai pēc tam pašā valodā. Tādējādi valodas formālais apraksts var iedvesmot tās izteiksmīguma attīstību.

1.3. Saistīti darbi

Šablonu realizācijas problēmai ir veltīts plašs šablonu literatūras klāsts. Ir vispāratzīts, ka nevar attēlot šablonus kodā, bet tikai nodrošināt konkrētu realizāciju (*Alexandrescu*, 2001). Izplatītu projektēšanas šablonu realizācija dažādās valodās tiek aprakstīta atsevišķās grāmatās (piemēram, *C#* (*Bishop*, 2008) vai *Scala* (*Løkke*, 2009)). Šis fakts liecina, ka projektēšanas šablonu realizācija var būt ļoti netrivāla, kas šablonu kopienā ir atzīts jau ilgu laiku (*Buschmann et al.*, 2007b; *Sommerlad*, 2007).

Ir piedāvātas vairākas pieejas realizācijas sarežģītības risināšanai. Piemēram, Franks Bušmans u. c. mēģināja izveidot konfigurējamas ģeneriskas implementācijas, bet ātri parādīja, ka tas nav iespējams pat salīdzinoši vienkāršos gadījumos (*Buschmann et al.*, 2007b). Dažās pieejās tiek lietotas uzlabotas valodu iespējas, piemēram, *C++* veidņu metaprogrammēšana (angl. *template metaprogramming*) (*Alexandrescu*, 2001) vai aspektorientētās valodas (*Kiczales et al.*, 1997; *Hannemann and Kiczales*, 2002; *Monteiro and Gomes*, 2013). Pavols Bača un Valentino Vraničs piedāvāja līdzīgu ideju aizstāt plaši pazīstamos objektorientētos projektēšanas šablonus ar aspektorientētajiem (*Bača and Vranić*, 2011). Šķiet, ka nesaistīto pienākumu nodalīšana ir ļoti svarīga šablonu realizācijā, bet prasība izmantot ar to mērķi aspektorientēto valodu šķiet lieka.

Vēl viens pētījumu virziens ir šablonu dekompozīcija. Atklāto šablonu milzīgais skaits liecina, ka tie var sastāvēt no pamata celtniecības blokiem. Tā, *Uwe Cduns* un *Pariss Avgeriu* mēģināja identificēt šablonu arhitektūras primitīvus (*Zdun and Avgeriou*, 2008). Frančeska Arčelli Fontana u. c. aprakstīja izplatītu projektēšanas šablonu mikrostruktūras, lai atvieglotu šablonu atrašanu esošajās sistēmās (*Fontana et al.*, 2011, 2013). Jans Bošs aprakstīja projektēšanas šablonu modeļus kā slāņu un deleģēšanu salikšanu un piedāvāja tā saukto slāņoto objektu modeli (angl. *Layered Object Model*) izmantojošo realizāciju (*Bosch*, 1998). Neraugoties uz minētajiem mēģinājumiem, ierobežots universālo celtniecības bloku komplekts nav zināms un šķiet neiespējams.

Realizācijas sarežģītības problēmu var atrisināt arī no otra gala: paaugstinot programmēšanas valodu abstrakcijas līmeni un tieši iekļaujot šablonus valodās. Džozefs Gils un Deivids H. Lorencs aprakstīja projektēšanas šablonu pakāpenisku iekļaušanu programmēšanas valodās (*Gil and Lorenz*, 1998). Tomēr nav šaubu, ka daudzi šabloni ir pārāk sarežģīti un ģeneratīvi, lai tos varētu tieši atbalstīt programmēšanas valodās, un robeža starp šabloniem, kas var un nevar būt valodas daļa, ir neskaidra. Tāpēc šablonu realizācijas problēma joprojām pastāv un ir nepieciešami jauni risinājumu priekšlikumi.

Runājot par to, kā piedāvāto tipu sistēmu varētu uzbūvēt, jāņem vērā struktūras un uzvedības aspekti. Runājot par strukturālo pusi, vienkārši salikto datu tipu veidošanas veidi ir zināmi no tipu teorijas, piemēram, pāri, korteži, ieraksti (ieskaitot klases), saraksti, varianti (ieskaitot opcijas un uzskaitījumus) un atsauces (*Pierce*, 2002). Tomēr šie primitīvi ne vienmēr atspoguļo programmu idejas (piemēram, asociatīvais masīvs netiek uzskatīts par datu salikšanas primitīvu, lai gan tas ir atbalstīts dažās valodās). Strukturālie šabloni, ko aprakstīja Ērihs Gamma *et al.* (*Gamma et al.*, 1995), gluži pretēji, pārstāv augstāku abstrakcijas līmeni. Tādi šabloni kā adapteris (angl. *Adapter*), starpniekserveris (angl. *Proxy*) vai fasāde (angl. *Façade*) nav piemērojami valodas līdzekļi, jo tie apraksta konkrētus sarežģītās mijiedarbības starp vairākiem dalībniekiem gadījumus. Tādējādi ir nepieciešams vidējais abstrakcijas līmenis.

Promocijas darbā mēģināts rast līdzsvaru programmēšanas valodas agnostiskos datu struktūru aprakstīšanas veidos. Šādu veidu piemēri ir vienotā modelēšanas valoda (angl. *Unified Modeling Language*, *UML*) (*OMG*, 2017) un datu pārraides attēlojumi (*Zimmermann et al.*, 2017).

Galvenā uzvedības abstrakcija tipu teorijā ir lambda izteiksme (angl. *lambda expression*). Tā ir funkcija, ko var definēt vienreiz un pēc tam lietot citos programmas fragmentos aprēķinātiem argumentiem. Lambda izteiksmes jēdziens radies no Alonzo Čerča mēģinājuma formalizēt algoritma jēdzienu (*Church*, 1936). Čerča lambda rēķini nav vienīgais iespējamais variants, bet viens no populārākajiem programmu uzvedību formalizēšanas veidiem (*Pierce*, 2002).

Vēl viena spēcīga abstrakcija, kas ir īpaši populāra funkcionālās programmēšanas pētījumos, ir monāde (angl. *monad*). Monādes attēlo tādas "netīras" (angl. *impure*) iespējas kā stāvoklis, izņēmumi un turpinājumi tīri funkcionālajā vidē (*Wadler*, 1992b). Monādi definē tipa operators un trīs funkcijas, kurām ir jāievēro noteikti likumi (*Wadler*, 1992a).

Procedurālajās valodās uzvedība galvenokārt ietver vārda un vērtības asociācijas maiņu (piešķiri). Šim nolūkam tipu teorija apraksta tādus tipus kā `Ref` (atsauce, maināmu vērtību saturošas vietas abstrakcija), `Source` (atsauce, ko var izmantot tikai vērtības iegūšanai) un `Sink` (atsauce, kas var tikai izmantot vērtības piešķirei) (*Pierce*, 2002). Programmēšanas valodas tieši vai netieši izmanto šos tipus, lai realizētu stāvokļa izmaiņas.

Tomēr ir labi zināms, ka piešķires operatoram programmēšanas valodās var būt dažādas nozīmes. Piemēram, strukturālās operatīvās semantikas (angl. *Structural Operational Semantics*, *SOS*) ietvarā ir nošķirta piešķiršana (vērtību kopēšana) un saistīšana (atsauces piešķiršana) (*Mosses*, 2006). *Anzen* programmēšanas valodas autori cīnījās ar piešķiršanas neskaidrību, ieviešot trīs dažādus piešķires operatorus, lai skaidrotu piešķires semantiku (*Racordon and Buchs*, 2019). Piešķiri var ierobežot tipu modifikatori, piemēram, `const`

atslēgvārds *C++* valodā. Tipu modifikatori ir īpaša veida apakštipa attiecības (*Foster et al.*, 1999). Tipu modifikatori bieži aizliedz noteiktas darbības, taču tiem var būt arī cita semantika (*Carlson and Wyk*, 2019). Piešķires ierobežojumu galīgā forma ir vienīgās statiskās piešķires (angl. *single static assignment*, *SSA*) forma: katrs simbols programmā tiek piešķirts tikai vienu reizi (*Cytron et al.*, 1991). Lai gan tas skan tīri funkcionāli, imperatīvo valodu kompilatori (piemēram, *LLVM* balstīti (*Lattner and Adve*, 2004)) var izmantot *SSA* formu kā starpposma attēlojumu.

1.4. Pētījuma metodoloģija

Promocijas darbs metodoloģiski ir balstīts uz projektēšanas šabloniem un tipu teoriju. Projektēšanas šablonu loma ir divējāda. No vienas puses, labi zināmu projektēšanas šablonu dekompozīcija tiek izmantota, lai identificētu iespējas, kas atvieglotu to īstenošanu. Tādējādi zināmi projektēšanas šabloni ir ideju avots iespējamiem programmēšanas valodu attīstības virzieniem. No otras puses, pašas piedāvātās iespējas ir aprakstītas šablonu veidā. Šajā ziņā šablonu pieeja ir promocijas darba pamatmetodoloģija – tipu sistēmas iespējas nav izvēlētas patvaļīgi, bet gan sistemātiski aprakstītas, vispārinātas un analizētas šablonu veidā.

Tipu teorija tiek izmantota, lai formalizētu piedāvātos šablonus kā valodas līmeņa konstrukcijas. Identificētie šabloni tiek formalizēti kā tipi ar atbilstošu semantiku un darbībām. Šie tipi veido galveno promocijas darba rezultātu.

2. TENDENCES UN MĒRĶI

Šajā nodaļā tiek aplūkoti izstrādātās tipu sistēmas mērķi, kas izriet no objektorientēto valodu evolūcijas tendencēm un potenciālām projektēšanas šablonu realizācijai noderīgām iespējām, ko autors identificējis savā maģistra darbā. Šie mērķi galvenokārt tiek apspriesti statiski tipētu valodu kontekstā. Savā ziņā piedāvātās iespējas varētu palielināt statisko valodu elastību dinamisko valodu virzienā, taču nepārkāpjot tipēšanas noteikumus.

2.1. Statisko un dinamisko programmēšanas valodu konverģence

Tradicionālā atšķirība starp statiski un dinamiski tipētajām valodām kļūst mazāk skaidra. Pirmkārt, katra mainīgā anotēšana ar tipu statiski tipētajās valodās ir nogurdinoša. Mūsdienās daudzas statiski tipētās programmēšanas valodas ļauj izlaist tipa anotācijas, ja kompilators tās var secināt automātiski. Tipu secināšana (angl. *type inference*) nemaina statiski tipēto valodu raksturu, bet noņem programmētāja mainīgo anotēšanas slogu.

Otrkārt, statiski tipētajām programmēšanas valodām vēsturiski bija tendence dzēst izpildlaika tipu informāciju, bet tas ne vienmēr ir iespējams. Polimorfās funkcijas tiek atrisinātas izpildlaikā, un tādējādi ir jāsaglabā tipu informācija līdz šim brīdim (*Pierce*, 2002). *C#* veica loģisku nākamo soli un ieviesa dinamiskās klases, kas darbojas kā klases dinamiski tipētajās valodās (*ECMA-334*, 2022).

No otras puses, dinamiski tipētās valodas iegūst iespējas kodā norādīt simbolu tipus. *PHP* ļauj noteikt tipu specifikācijas funkciju parametriem un atgriešanas vērtībām kopš versijas 7.0 (*PHP*, 2023). Iespēja norādīt funkciju parametru tipus tika ieviesta arī *Python 3.12* (*Pyt*, 2023). Savukārt *TypeScript* ir vesela valoda, kas atbalsta *JavaScript* mainīgo un funkciju kompilēšanas laika tipa deklarācijas (*Mic*, 2023).

Tādējādi, lai gan statiski un dinamiski tipētās valodas savos pamatprincipos joprojām atšķiras, tās pakāpeniski iegūst iespējas, kas ļauj izmantot pieejas, kas parasti ir saistītas ar otru klasi. Lai atbalstītu konverģences tendenci, promocijas darba 2.1. apakšnodaļā formulēts vispārinājums, kas aptver gan tradicionālās statiski tipētās, gan dinamiski tipētās pieejas kā atsevišķus gadījumus.

2.2. Objektorientēto un funkcionālo valodu konverģence

Funkcionālā programmēšana ir programmēšanas paradigma, kuras pamatā ir stingri pieņēmumi, piemēram, piešķires, maināmu datu un iterācijas neesamība. Piešķires vietā funkcionālās programmas rada jaunas vērtības; datu struktūras maiņas vietā tās rada citu struktūru, kas var daļēji koplietot datus; iterācijas vietā tās izmanto rekursiju (*Michaelson*, 2011). Neskatoties uz ierobežojošo vidi, funkcionālā programmēšana atrisina tādas pašas uzdevumu klasi kā imperatīvā programmēšana un citas programmēšanas paradigmas.

Pēdējā laikā pieaug funkcionālās programmēšanas popularitāte. Tā rezultātā objektorientētās programmēšanas valodas ir ieguvušas noteiktas iespējas, kas agrāk bija cieši saistītas tikai ar funkcionālo programmēšanu, piemēram, nemaināmus datu tipus (angl.

immutable data types) un augstākas kārtas funkcijas (angl. *higher-order functions*). Galvenās valodas aizņem tikai dažas funkcionālās iespējas, taču tādas valodas kā *Scala* un *Swift* pilnībā iekļauj funkcionālo stilu (lai gan tās joprojām nav "tīras", jo atbalsta arī piešķiri un maināmos datus).

Tradicionāli funkcionālās programmēšanas priekšrocības ir saistītas ar matemātisko pamatotību. Parastās matemātiskās sistēmas darbojas ar vispārpieņemtām patiesībām, nevis maināmo stāvokli un darbību secību. Teorētiski funkcionālās programmas korektums ir teorēma, ko var pierādīt vai apgāzt. Praksē korektuma pierādīšana ir sarežģīta pat funkcionālajai programmai (*Michaelson*, 2011). Tomēr vēsturisku iemeslu dēļ funkcionālās valodas veidotāji daudz lielāku uzmanību pievērš tipu sistēmu garantijām un to formālajai pierādīšanai.

Vēl viena funkcionālo valodu priekšrocība ir saistīta ar to neatkarību no izpildes secības. Pateicoties šai neatkarībai, funkcionālās programmas pēc būtības ir drošas laiksakritīgajā vidē. Nav nepieciešami bloķēšana vai citi sinhronizācijas mehānismi, jo nav kopīga maināma stāvokļa (*shared mutable state*).

Funkcionālās programmēšanas ierobežojumi rada arī dažus trūkumus. Pirmkārt, rekursija bieži ir mazāk efektīva nekā iterācija. Bieži ir iespējams pārrakstīt rekursīvo algoritmu astes rekursīvajā veidā, kas ļauj kompilatoram optimizēt bināro kodu (*Pratt and Zelkowitz*, 2001), taču šī forma ir sarežģītāka un zaudē rekursīvās definīcijas galveno priekšrocību – dabisko atbilstību starp funkcijas definīciju programmā un matemātisko definīciju. Vēl viens neefektivitātes iemesls ir saistīto datu struktūru, piemēram, sarakstu, ciešana no neefektīvas atmiņas piekļuves pārtraukti izvietotiem datiem.

Vēl viens trūkums ir tas, ka ne visas plaši izmantotās datu struktūras pieļauj rekursīvas definīcijas. Piemēram, virknes, kas atbalsta efektīvu brīvpiekļuvi saviem elementiem (piemēram, masīvus imperatīvajās valodās), ir grūti definēt rekursīvi. Funkcionālie algoritmi tajos var būt sarežģīti, jo algoritma rekursīvā struktūra parasti atspoguļo pašas datu struktūras rekursīvo struktūru (*Michaelson*, 2011).

Neskatoties uz aprakstītajām problēmām, funkcionālo programmēšanas valodu iespējas ir cieši saistītas ar matemātisko pamatotību un pavedienu drošumu. Ievērojot noteiktus noteikumus, programmēšana funkcionālajā stilā ir iespējama arī valodās, kas nav tīri funkcionālas.

Galvenās iespējas, kas jāņem vērā valodas līmenī, ir funkcijas kā vērtības (4.1. apakšnodaļa) un algebriskie datu tipi (3.6. un 3.8. apakšnodaļa). Citas svarīgas valodas līmeņa konstrukcijas ir garantētais nemainīgums un parametriskais polimorfisms, taču promocijas darbā tie netiek aplūkoti, lai saglabātu diskusijas fokusu.

2.3. Tradicionālo objektorientēto ierobežojumu mīkstināšana

Dažu objektorientētu programmēšanas valodu jēdzienu un konstrukciju lietošana tradicionāli ir pakļauta ierobežojumiem drošuma vai laba objektorientēta projektēšanas dēļ. Tomēr daži no šiem ierobežojumiem laika gaitā mēdz tikt mīkstināti (*Batdalov*, 2017). Tas parāda, ka sākotnējās konstrukcijas bija pārāk ierobežojošas un prasīja paplašinājumu labākam izteiksmīgumam.

Šāda procesa piemērs ir saskarnes (angl. *interface*, arī piejaucamais (angl. *mixin*) vai iezīme (angl. *trait*)) definēšanas paplašinājums. Klasiskajā *Java* veidā saskarnēs varēja būt tikai publisko metožu deklarācijas. Tomēr kopš *Java 8* saskarnes var definēt arī tā saukto "noklusējuma" metožu realizāciju (*Gosling et al.*, 2023). *TypeScript* valodā saskarnes var definēt arī datu locekļus (*Mic*, 2023). *Scala* valodā saskarnes var definēt privātos locekļus. *Scala* saskarnēs ir aizliegti tikai konstruktora parametri (*Odersky et al.*, 2023). Tādējādi ir tendence paplašināt saskarņu iespējas.

Vēl viens piemērs ir saistīts ar datu piekļuves ierobežojumu attīstību. Visu datu locekļu padarīšana par privātiem un piekļuve tiem tikai caur metodēm tiek uzskatīta par labu stilu, izņemot klasēm, kurām ir apzināti atvērta iekšējā struktūra (*Martin*, 2008). Tomēr, to izdarījuši, programmētāji bieži izveido ieguvējus (angl. *getter*) un iestatītājus (angl. *setter*), lai piekļūtu privātajiem laukiem. Lai gan vajadzētu izvairīties no ieguvēju un iestatītāju izveides katram privāto datu loceklim (*Martin*, 2008), šī darbība ir tik tipiska, ka tā atdzīvina tādas konstrukcijas kā īpašības *C#* valodā (*ECMA-334*, 2022) un iegūšanas un iestatīšanas metodes *TypeScript* valodā (*Mic*, 2023). Atkal sākotnējie ierobežojumi bija pārāk stingri un prasīja lielāku elastību.

Vēl viens piemērs ir atsaucēm līdzīgu tipu dažādība *C++* un *Java*. *C++* sākotnēji atbalstīja no *C* mantotus rādītājus (angl. *pointers*), kā arī atsauces (angl. *references*), to drošuma nolūkos ierobežoto versiju. Atsauces risināja dažus rādītāju trūkumus, bet bija nepiemērojamas citās situācijās, kā rezultātā radās viedie rādītāji, kas nodrošina citus drošuma aspektus (*ISO/IEC 14882:2020*, 2020). Līdzīgi, papildus iebūvētajiem *Java* atsauces tipiem (masīviem un objektiem), standarta bibliotēkā ir tādi tipi kā `SoftReference`, `WeakReference` un `PhantomReference`, kuriem ir īpaši drazu savākšanas noteikumi (*Ora*, 2023). Atmiņas pārvaldību nav iespējams reducēt līdz vienam vai diviem pamata tipiem.

Vissvarīgākā mācība no minētajiem piemēriem ir tāda, ka nav jēgas definēt divus vai vairākus līdzīgus jēdzienus, kas atšķiras vairāk nekā vienā aspektā. Andrejs Aleksandresku rakstīja, ka katram neatkarīgam projektēšanas lēmumam vai ierobežojumam jākļūst par atsevišķu politiku un jāatbalsta katra ortogonālo politiku kombinācija (*Alexandrescu*, 2001). Šo principu autors centās pēc iespējas ievērot aplūkotajā tipu sistēmā.

2.4. Operatori kā klasē definētas metodes

Šī tendence ir vāji saistīta ar promocijas darbu, jo tipu sistēmām ir mazs sakars ar operatoriem. Operatori galvenokārt ir sintaktiskas konstrukcijas, kas ietekmē to, kā kompilators veido abstrakto sintakses koku. Tāpēc promocijas darbā operatori ir apskatīti tikai īsi.

Parasti programmēšanas valodai ir fiksēts operatoru kopums, un valoda nodrošina iebūvēto tipu operatoru ieviešanu. Runājot par lietotāja definētiem tipiem, dažas valodas ļauj programmētājiem definēt operatorus šiem tipiem (piemēram, *C++* (*ISO/IEC 14882:2020*, 2020)), bet dažas neļauj (piemēram, *Java* (*Gosling et al.*, 2023)). *Scala* valodai ir divas jaunas iespējas – patvaļīga operatora definēšana un izsaukuma stratēģija operatoros (tāpat kā citās metodēs) (*Odersky et al.*, 2023). Pēdējā stratēģija nav unikāla, bet citās valodās to parasti izmanto tikai iebūvētajos operatoros.

Apskatīto tendenci var vispārināt šādi: katru iespēju vai pazīmi, ko var attiecināt uz iebūvēto tipu, var attiecināt arī uz lietotāja definētiem tipiem. Tas ir viens no galvenajiem iemesliem vispārinātu funkciju ieviešanai (4.1. apakšnodaļa).

2.5. Izplatītiem projektēšanas šabloniem atbilstošas konstrukcijas

Attiecības starp projektēšanas šabloniem un valodas konstrukcijām nav triviālas. Ērihs Gamma *et al.* teica, ka projektēšanas šablons vienā valodā ir tikai valodas konstrukcija citā (*Gamma et al.*, 1995). Tas attiecas arī uz valodu attīstību laika gaitā – daudzas programmēšanas valodas ir ieguvušas izplatītiem projektēšanas šabloniem atbilstošas iespējas. Šādu iespēju piemēri ir *for-each* cikls, iteratora (angl. *Iterator*) projektēšanas šablona (*Gamma et al.*, 1995) realizācija, objekti *Scala* (*Odersky et al.*, 2023), vienīgā (angl. *Singleton*) projektēšanas šablona (Gamma *et al.*, 1995) realizācija, kanāli *Go* (*Goo*, 2023), cauruļu un filtru (angl. *Pipes and Filters*) arhitektūras šablona (*Buschmann et al.*, 2013) realizācija, un notikumi *C#* (*ECMA-334*, 2022) un novērojamie (angl. *observables*) *Kotlin* (*Jet*, 2023), izdevēja-abonenta (angl. *Publisher-Subscriber*) projektēšanas šablona (*Buschmann et al.*, 2013) realizācija.

Promocijas darbs seko tai pašai tendencei, aprakstot šablonus, kas potenciāli var būt programmēšanas valodas iespējas.

2.6. Vispārināšana un saskarnes izvilkšana

Pārmantošana objektorientētās programmēšanas valodās ir atsevišķs apakštipēšanas gadījums, jo jebkurš apakšklases objekts tiek uzskatīts par tās virsklases objektu (*Pierce*, 2002). Daži autori kritizēja apakštipu interpretāciju, tā vietā argumentējot par vispārēju pakāpenisko modifikāciju (*Cook et al.*, 1989; *Taivalsaari*, 1996). Tomēr lielākā daļa valodu joprojām pieņem apakštipu attiecības starp klasēm un apakšklasēm.

Autors apgalvoja, ka iespēja definēt virsklasi vēlāk nekā apakšklasi palīdzētu aptvert plašāku pakāpenisko modifikāciju klāstu, kā arī atvieglotu projektēšanas šablonu realizāciju (*Batdalov*, 2010; *Batdalov and Nikiforova*, 2016). Tradicionālais objektorientētais tipu hierarhijas veidošanas veids ir deduktīvs – no vispārīgā uz atsevišķo vai no virstipa uz apakštipu. Tomēr izziņas process bieži notiek pretējā virzienā. Piemēram, vispārīgākie kompleksie skaitļi vēsturiski tika ieviesti vēlāk nekā reālo skaitļu atsevišķs gadījums.

Definīcijas secība nav tieši saistīta ar tipu sistēmu, jo pēdējā attiecas uz jau definētiem tipiem. Tomēr promocijas darba 2.6. apakšnodaļā ir definēti noteikumi, kas jāievēro, lai klašu hierarhijās varētu veikt vispārināšanu specializācijas vietā.

2.7. Predikātu un dziļuma apakšklasēšana

Lielākā daļa galveno programmēšanas valodu atbalsta tikai platuma apakšklasēšanu (angl. *width subclassing*), t. i., jaunu locekļu pievienošanu klasei. Tipu teorija apraksta arī dziļuma apakšklasēšanu (angl. *depth subclassing*), t. i., esošo locekļu tipu apakštipēšanu (*Pierce*, 2002). Dziļuma apakšklasēšanu var vispārināt līdz predikātu apakštipēšanu (angl. *predicate*

subtyping), kurā apakštipu var sašaurināt, izmantojot patvaļīgu predikātu (piemēram, noteiktam formātam atbilstošo virkni) (*Batdalov and Nikiforova*, 2016).

Autors iepriekš parādīja, ka dziļuma un predikātu apakšklasēšana būtu izdevīga vispārzināmu projektēšanas šablonu realizācijai (*Batdalov and Nikiforova*, 2016). Tomēr šāda veida apakšklasēšana rada grūtības, izmantojot apakšklases atsauces kā maināmas virsklases atsauces. Platuma apakšklasēšana garantē, ka šāda izmantošana nepārkāps tipu drošuma noteikumus, bet dziļuma apakšklasēšana un predikātu apakštipēšana to negarantē.

Promocijas darba 2.7. apakšnodaļā formulēti noteikumi, kas ļautu drošu dziļumu un predikātu apakšklasēšanu. Tie ir balstīti lietošanas vietas variantumā (angl. *usage-site variance*) tipēšanas noteikumiem ģeneriskajiem (parameterizētajiem) tipiem.

2.8. Noklusējuma realizācija un izpildes konteksti

Parasti, lai izveidotu objektu, ir jānorāda konkrēta klase, kas jāveido. Nepietiek norādīt tikai abstraktu klasi. Autors apgalvoja, ka pārpiesaistāma abstraktu klašu noklusējuma realizācija (t. i., abstraktās klases instantiēšanai lietojama konkrētā klase) būtu izdevīgas vispārzināmu projektēšanas šablonu realizācijā (*Batdalov and Nikiforova*, 2016). Līdzīga pieeja tiek izmantota *Scala*, kur abstraktās klases pavadošais objekts var radīt konkrētu realizāciju (*Odersky et al.*, 2023). Tomēr *Scala* noklusējuma realizācija ir iestatīta bibliotēkā, un programmētājs to nevar mainīt.

Noklusējuma realizācijas piešķire klasei ir piesaistīšanas veids, tāpēc tās darbības joma ir skaidri jādefinē (*Batdalov and Nikiforova*, 2016). Promocijas darbs piedāvā šo piesaistīšanu saglabāt izpildes kontekstos. Izpildes konteksts ir globāli pieejamu nosaukumu telpa, kas tiek mantota, veidojot jaunu pavedienu. Izpildes konteksti ir saistīti ar izpildītājiem (4.2. apakšnodaļa).

2.9. Hameleona objekti

Stāvokļa (angl. *State*) šablons ļauj objektam mainīt savu uzvedību izpildlaikā tā, ka šķiet, ka tas maina savu klasi (*Gamma et al.*, 1995). Stāvokļa šablons ir atspoguļots *UML* stāvokļa mašīnas diagrammā, lai gan *UML* pieņem, ka visu stāvokļu uzvedība ir realizēta vienā klasē (*OMG*, 2017). Tomēr nesaistītas uzvedības ievietošana dažādās klasēs ir labāks kodēšanas stils (*Martin*, 2008). Tāpēc promocijas darbā aplūkota daudzklašu stāvokļa mašīnas realizācija.

Sākotnējā stāvokļa šablona realizācija izmanto papildu novirzīšanas līmeni, kas novirza izsaukumus uz faktiskiem objektiem (*Gamma et al.*, 1995). Autors piedāvāja atbalstīt šādu uzvedību tieši valodas līmenī, ļaujot objektiem mainīt savas klases izpildlaikā (*Batdalov and Nikiforova*, 2016).

Promocijas darba 2.9. apakšnodaļā formulēti noteikumi, kas nepieciešami šīs pieejas tipu drošumam. Tie ir balstīti pieņēmumā, ka klases mainošas metodes var mainīt izpildlaika tipu, bet tipa pārbaudes aizliedz izsaukt metodi, ja jaunais izpildlaika tips nav mainīgā kompilēšanas laika tipa apakštips.

2.10. Paplašināta inicializācija

Valodai var būt atšķirīgi noteikumi dažādām objekta dzīves cikla fāzēm, piemēram, tikai lasāmu lauku piešķire ir iespējama tikai izveides laikā. Rezultātā visiem objekta inicializācijai nepieciešamajiem datiem ir jābūt viena konstruktora parametriem. Šo ierobežojumu ir iespējams daļēji pārvarēt, izmantojot izveidošanas šablonus, piemēram, būvētāju (angl. *Builder*) un rūpnīcas metodi (angl. *Factory Method*) (*Gamma et al.*, 1995). Tomēr tiem nav īpaša statusa, un tie neļauj modificēt konstantos laukus ārpus konstruktora.

Autors piedāvāja pagarināt inicializācijas fāzi un ļaut tai ietvert vairākas darbības (*Batdalov and Nikiforova*, 2016). Tas būtībā nozīmē dažādu stāvokļu esamību dažādām dzīves cikla fāzēm. Tādējādi priekšlikumu var īstenot, apvienojot 2.7. un 2.9. apakšnodaļā aprakstītās pieejas.

2.11. Objektu mijiedarbības stili

Tipiskākais objektu mijiedarbības veids ir sinhronie funkciju izsaukumi. Tomēr programmēšanas valodas mēdz ieviest alternatīvus mijiedarbības veidus, piemēram, asinhronos izsaukumus daudzās valodās un kanālos *Go* valodā (*Batdalov*, 2017). Šie veidi ir reducējami līdz sinhroniem izsaukumiem, bet ļauj dabiskāk izteikt objektu mijiedarbību.

Tomēr ir iespējami arī citi mijiedarbības stili. Klase var attēlot ārēju sistēmu vai komponentu, piemēram, ja tiek izmantots fasādes (angl. *Façade*), starpniekservera (angl. *Proxy*) (*Gamma et al.*, 1995) vai brokera (angl. *Broker*) (*Buschmann et al.*, 2013) šablons, tāpēc mijiedarbība ar klasi nozīmē mijiedarbību ar šo ārējo sistēmu. Komponentu mijiedarbības stili ir daudzveidīgāki: sinhronais pieprasījums-atbilde, asinhronais pieprasījums-atbilde, caurule un filtrs, pārraide, tāfele un publicēšana-abonēšana (*Crnković et al.,* 2011). Ir aprakstīti daudzi šos stilus pārstāvoši projektēšanas šabloni: novērotājs (angl. *Observer*) (*Gamma et al.*, 1995), tāfele (angl. *Blackboard*), pārsūtītājs-saņēmējs (angl. *Forwarder-Receiver*), kungs-vergs (angl. *Master-Slave*), caurules un filtri (angl. *Pipes and Filters*), starpniekserveris (angl. *Proxy*), izdevējs-abonents (angl. *Publisher-Subscriber*) (*Buschmann et al.*, 2013).

Nav iespējams atbalstīt katru mijiedarbības stilu tieši valodas līmenī, jo mijiedarbības stilu saraksts ir potenciāli atvērts. Tā vietā konkrētie mijiedarbības stili ir jādefinē kā bibliotēkas klases un operatori. Aplūkojamajā tipu sistēmā atbalstošas iespējas ir vispārinātas funkcijas (4.1. apakšnodaļa), izpildītāji (4.2. apakšnodala), funkcijas, kas vairākas vērtības atgriež pa vienai (4.4. apakšnodaļa), un izpakošanas piešķire (5.5. apakšnodaļa).

2.12. Apstrādes stāvokļa vadība

Parastās funkcijas programmēšanas valodās un tipu teorijā ir tikai uzvediskas; tajā ir tikai kods un nav nekāda stāvokļa. Komandas (angl. *Command*) šablons vispārina šo koncepciju kā pirmās klases objektu, kas pieļauj pārmantošanu, var saglabāt izpildes stāvokli un atbalsta atcelšanu, pārtaisīšanu un reģistrēšanu (*Gamma et al.*, 1995). Franks Bušmans u. c. pamanīja, ka visas šīs funkcionalitātes realizācija komandā var būt nepamatota, un aprakstīja komandu

procesora (angl. *Command Processor*) šablonu (*Buschmann et al.*, 2013). Komandu procesors ir izpildītāja abstrakcija, piemēram, pavediens, pavedienu pūls vai atkļūdošanas vide.

Aplūkotā tipu sistēma lieto šīs abstrakcijas kā pamatelementus. Atbilstošās tipu sistēmas iespējas ir vispārinātas funkcijas (4.1. apakšnodaļa) un izpildītāji (4.2. apakšnodaļa).

3. DATU SALIKŠANA

Datu salikšana tiek lietota programmēšanas valodās, lai izveidotu saliktus datu tipus (piemēram, masīvus un klases) no vienkāršākiem (*Pierce*, 2002). Līdzīgi mehānismi tiek lietoti datu pārraides attēlojuma valodās (piemēram, *JSON*, *XML* un tīkla protokolos). Tomēr konkrētie šajās valodās izmantotie mehānismi ir atšķirīgi, kas var radīt grūtības, pārveidojot starp datu pārraides attēlojumu un atmiņā esošajiem objektiem.

Datu pārraides attēlojums (angl. *data transfer representation*) ir attēlojums, kas ir agnostisks pret datu interpretēšanai izmantotajām programmēšanas valodām un paradigmām (*Zimmermann et al.*, 2017). Tāpēc datu pārraides attēlojuma valodas ir labs izplatītu no konkrētām programmēšanas valodām neatkarīgu šablonu avots. Datu objekta programmēšanas valodā pārveidošanu tā datu pārraides attēlojumā sauc par datu pakošanu (angl. *marshalling*), un pretējā darbība ir izpakošana (angl. *unmarshalling*) (*Zimmermann et al.*, 2017). Izpakošanai parasti ir nepieciešama papildu informācija (shēma), lai saprastu, kā interpretēt datus.

Šajā nodaļā tiek aprakstīti salikšanas primitīvi, kas tiek izmantoti gan programmēšanas valodās, gan datu pārraides attēlojumā. Šo primitīvu pamatā ir autora piedāvātais salikto pamattipu saraksts programmēšanas valodas iespēju salīdzināšanai (*Batdalov*, 2017; *Batdalov et al.*, 2016), kas vēlāk tika aprakstīts šablonu veidā (*Batdalov and Nikiforova*, 2018, 2019).

3.1. Virkne

Virkne (angl. *Sequence*) ir krājums, kas ietver potenciāli atkārtojošas noteiktā kārtībā sakārtotas vērtības, ja kārtība vispārējā gadījumā nav atkarīga no pašām vērtībām (piemēram, vērtībām nav jābūt augošā vai dilstošā kārtībā). Vērtībām var piekļūt secīgi norādītajā kārtībā vai pēc indeksa virknē. Virkne var atbalstīt izmēru maiņu, lai saglabātu vairāk vai mazāk vērtību. Vērtībām virknē var būt viens un tas pats tips vai dažādi.

Virkņu piemēri programmēšanas valodās ietver tādus izplatītus datu tipus kā fiksēta un mainīga izmēra masīvi, vienreiz un divreiz saistītie saraksti un korteži. Šie pamattipi var kalpot augstāka līmeņa virkņu, piemēram, steku un rindu, realizācijai. Datu pārraides attēlojuma valodās virknes parasti attēlo ar vērtību virknēm (masīva literāļiem *JSON* vai atkārtotiem laukiem *XML*).

Izmantotā formalizācija virkni definē kā universālo (ģenerisko) eksistenciālo (abstrakto) tipu, kas ļauj izveidot virkni no vienreiz pārejamā (4.4. apakšnodaļa), iterēt pa vērtībām un iterēt pa piešķiramajām ligzdām, kas ļauj mainīt virknes elementus. Citas darbības, tostarp piekļuve pēc indeksa, nav kopīgas visām virknēm, tāpēc tām jāpieder pie konkrētākiem tipiem.

3.2. Kopa

Kopa (angl. *Set*) ir unikālu vērtību krājums, kuru kārtība nav definēta vai ir atkarīga no pašām vērtībām (t. i., sakārtotā kopa var iterēt pa vērtībām augošā kārtībā). Atšķirībā no virknēm kopai nav no vērtībām neatkarīgas kārtības. Kopa var atbalstīt pārbaudi, vai noteikta vērtība ir kopā. Kopā esošajām vērtībām var būt viens un tas pats tips vai dažādi.

Programmēšanas valodas bieži atbalsta sakārtotās un nesakārtotās kopas, kuru pamatā parasti ir attiecīgi sarkanmelnie koki un jaucējtabulas. Ir iespējamas arī citas realizācijas. Dažās valodās nav atsevišķu tipu kopām, taču ir iespējams kopu attēlošanai izmantot kartes ar triviālām vērtībām. Datu pārraides attēlojuma valodās kopu attēlojums ir tāds pats kā virkņu, un datu interpretācija kā virknes vai kopas ir atkarīga no datu shēmas.

Izmantotā formalizācija kopu definē kā universālo (ģenerisko) eksistenciālo (abstrakto) tipu, kas ļauj izveidot kopu no vienreiz pārejamā (4.4. apakšnodaļa), iterēt pa vērtībām un pievienot un noņemt elementu. Izveidošanai un iterēšanai ir tā pati deklarācija kā virknēs, bet to semantika atšķiras, jo kopa nesaglabā vērtību ievietošanas kārtību. Pārbaude, vai vērtība ir kopā, nav iekļauta formalizācijā, jo dažās realizācijās šo darbību nevar veikt efektīvāk nekā ar iterēšanu (lai gan šādas realizācijas ir retas).

3.3. Daudzkopa

Daudzkopa (angl. *Multiset*) ir potenciāli atkārtojošos vērtību krājums, kuru kārtība nav definēta vai ir atkarīga no pašām vērtībām (t. i., sakārtotā daudzkopa var iterēt pa vērtībām augošā kārtībā). Tāpat kā kopās un atšķirībā no virknēm daudzkopām nav no vērtībām neatkarīgas kārtības. Daudzkopa var atbalstīt pārbaudi, vai noteikta vērtība kopā. Vērtībām daudzkopā var būt viens un tas pats tips vai dažādi.

Programmēšanas valodas reti atbalsta daudzkopas, taču tās ir viegli emulēt ar kartēm (kā vērtību saglabājot ietilpšanu skaitu). *C++* ir izņēmums, kas tieši atbalsta sakārtotās un nesakārtotās daudzkopas. Datu pārraides attēlojuma valodās daudzkopu attēlojums ir tāds pats kā virknēm un kopām, un datu interpretācija ir atkarīga no datu shēmas.

Izmantotā formalizācija daudzkopu definē kā universālo (ģenerisko) eksistenciālo (abstrakto) tipu, kas ļauj izveidot kopu no vienreiz pārejamā (4.4. apakšnodaļa), iterēt pa vērtībām un pievienot un noņemt elementu. Šī formalizācija ir tāda pati kā kopām; vienīgā atšķirība ir, vai mainošās darbības nodrošina vērtību unikalitāti.

3.4. Karte

Karte (anlg. *Map*) ir atbilstība starp atslēgām un vērtībām, kur katra atslēga var parādīties tikai vienu reizi. To var uzskatīt par atslēgu un vērtību pāru kopu, kur atslēgas neatkārtojas un tām nav no atslēgām neatkarīgas kārtības. Atslēgas var būt iepriekš definētas vai nē. Vērtībai parasti var piekļūt, izmantojot atslēgu. Kartei nav prasību vērtībām – tām nav jābūt unikālām, un nav nepieciešama efektīva piekļuve atslēgām pēc vērtībām.

Tipisks fiksēto atslēgu kartes piemērs ir ieraksts (angl. *record*). Mainīgo atslēgu karšu piemēri ietver asociatīvus masīvus, kas var būt sakārtoti (piemēram, pamatoti sarkanmelnos kokos) vai nē (piemēram, pamatoti jaucējtabulās). Atkarībā no valodas klašu pamatā ir ieraksts vai nesakārtotais asociatīvais masīvs. Attiecībā uz datu pārraides attēlojuma valodām, *JSON* atbalsta objektu literāļus un atslēgu un vērtību pāru masīvus, bet *XML* nav tieša kartes ekvivalenta (lai gan atsevišķus gadījumos var emulēt ar atkārtotiem tagiem).

Izmantotā formalizācija karti definē kā universālo (ģenerisko) eksistenciālo (abstrakto) tipu, kas ļauj izveidot karti no atslēgu un vērtību pāru vienreiz pārejamā (4.4. apakšnodaļa), iterēt pa atslēgu-vērtību pāriem, iterēt pa atslēgu un vērtībām piešķiramo ligzdu pāriem, kā arī pievienot un noņemt elementus. Līdzīgi kā kopās, piekļuves ar atslēgu darbība tiek izlaista, jo dažas (lai gan retas) realizācijas neatbalsta šo darbību efektīvāk nekā ar iterēšanu.

3.5. Daudzkarte

Daudzkarte (anlg. *Multimap*) ir atbilstība starp atslēgām un vērtībām, kur atslēgas var atkārtoties. To var uzskatīt par atslēgu-vērtību pāru kopu, kur atslēgas var atkārtoties un kurām nav no atslēgām neatkarīgas kārtības. Vērtībām parasti var piekļūt, izmantojot atslēgu. Vairākas vērtības, kas saistītas ar vienu un to pašu atslēgu, var būt sakārtotas virknē vai kopā. Daudzkartei nav prasību vērtībām – tām nav jābūt unikālām, un nav nepieciešama efektīva piekļuve atslēgām pēc vērtībām.

Programmēšanas valodas reti atbalsta daudzkartes, taču tās ir viegli emulēt ar kartēm, kuru vērtības ir kopas vai virknes. *C++* ir izņēmums, kas tieši atbalsta sakārtotās un nesakārtotās daudzkartes. Datu pārraides attēlojuma valodās *JSON* atbalsta atslēgu-vērtību pāru masīvus, bet *XML* nav tieša daudzkartes ekvivalenta (atsevišķu gadījumus var attēlot, taču nav vienota veida tam).

Izmantotā formalizācija daudzkarti definē kā universālo (ģenerisko) eksistenciālo (abstrakto) tipu, kas ļauj izveidot karti no atslēgu un vērtību pāru vienreiz pārejamā (4.4. apakšnodaļa), iterēt pa atslēgu-vērtību pāriem, iterēt pa atslēgu un vērtību krājumu pāriem, iterēt pa atslēgu un vērtībām piešķiramo ligzdu krājumu pāriem, kā arī pievienot un noņemt elementus.

3.6. Varianta tips

Variants (anlg. *Variant*) ir izvēle starp divām vai vairākām alternatīvām (piemēram, mainīgais, kurā var būt vesels skaitlis vai virknes vērtība; tipu apvienojums). Parasti variantiem ir jēga tikai deklarācijas vietā (kompilēšanas laika tips vai datu attēlojuma shēma), jo lietošanas vietā (izpildlaika tips vai faktiskais datu attēlojums) ir ietverta noteikta alternatīva. Variants var būt iezīmēts (ar izvēlēto alternatīvu norādošu iezīmi, angl. *labelled variant*) vai neiezīmēts (vienkāršs tipu apvienojums, angl. *unlabelled variant*). Svarīgi variantu atsevišķi gadījumi ir neobligātie (angl. *options*) un uzskaitījumi (angl. *enumerations*).

Variantu atbalsts dažādās valodās ievērojami atšķiras. Iezīmēto variantu piemēri programmēšanas valodās ietver `std::variant` *C++* valodā un `Either` *Scala* valodā. *TypeScript* atbalsta `|` tipu operatoru, kas nozīmē neiezīmētu variantu. Atsevišķi gadījumi tiek atbalstīti biežāk. Piemēram, valodās ar izņēmumiem funkcija netieši atgriež atgriešanas vērtības un izņēmuma variantu. *Scala* valodā šis varianta tips eksistē arī tieši (`Try`). Daudzas valodas atbalsta uzskaitījumus, triviālo tipu variantus, kur katrā tipā ir tikai viena vērtība. Neobligātā vērtība (vēl viens atsevišķs gadījums) ir apskatīta 3.7. apakšnodaļā. Datu pārraides attēlojuma valodās *XML* shēma ļauj deklarēt alternatīvas, ko var ievietot noteiktā dokumenta vietā.

Izmantotā formalizācija iezīmēto variantu definē kā universālo (ģenerisko) eksistenciālo (abstrakto) tipu, kas ļauj izveidot variantu no jebkuras alternatīvas vērtības, lietot dažādas alternatīvas akceptējošu funkciju kopu (lai tikai funkcija, kas pieņem turēto alternatīvu tiek izpildīts), un lietot funkciju kopu, kas pieņem piešķiramās ligzdas dažādām alternatīvām.

3.7. Neobligātā vērtība

Neobligātā vērtība (anlg. *Optional Value*) ir mainīgais, kuram var būt vai nebūt vērtība. Tas ir varianta atsevišķs gadījums – tipa un atdalītas nullvērtības variants. Līdzīgi kā vispārīgos variantos, neobligātā vērtība var būt iezīmēta vai neiezīmēta, lai gan atšķirība parādās tikai noteiktos malu gadījumos (iezīmētās neobligātās vērtības var ligzdot, piemēram, `Optional[Optional[Int]]`, bet neiezīmētās vērtības to neļauj).

Daudzās valodās atsauces tipi (rādītāji *C++* valodā, objekti un masīvi *Java* utt.) var ietvert īpašu nullvērtību, kas padara šos tipus neobligātus. Šie tipi ļauj atsauces izmantošanu, nepārbaudot vērtību, un mēģinājums izmantot nullatsauci rada izņēmumu. Šīs pieejas radītājs vēlāk to nosauca par miljardu dolāru kļūdu (*Hoare*, 2009). Valodas ar spēcīgu funkcionālu ietekmi atbalsta tipu drošu neobligāto tipu, kur nevar izmantot atsauci, nepārbaudot, vai vērtība ir klāt.

Izmantotā formalizācija neobligāto vērtību definē kā universālo (ģenerisko) eksistenciālo (abstrakto) tipu, kas ļauj izveidot tukšu vērtību, izveidot netukšu vērtību no pamatā esošā tipa vērtības un divas metodes (nemainošo un mainošo), kas izvēlas vienu no divām dotajām funkcijām atkarībā no tā, vai vērtība ir tukša.

3.8. Tipu šķēlums

Tipu šķēlums (angl. *type intersection*) dabiski atbilst daudzkārtējai pārmantošanai objektorientētās valodās, jo vairāku klašu apakšklase ir arī visu to apakštips. Tomēr objektorientētās valodās reti tiek atbalstīta daudzkārtēja pārmantošana no klasēm. Parasti tiek atbalstīta vairāku saskarņu mantošana, un tā nodrošina iespēju izveidot tipu šķēlums. Tā kā tipu šķēlums ir daudzkārtējas pārmantojuma dabiskas sekas, nav nepieciešams aprakstīt atsevišķu šablonu.

4. SKAITĻOŠANAS ELEMENTI

Šajā nodaļā aprakstīti pamata skaitļošanas šabloni, kas paredzēti programmas uzvedības modelēšanai. Tie neietver 5. nodaļā aprakstītos piešķires šablonus. Vienreiz pārejamā un vienreiz piešķiramā šabloni iepriekš tika aprakstīti kā programmas uzvedības un datu salikšanas sasaistes šabloni (*Batdalov and Nikiforova*, 2018); pārējie ir balstīti autora piedāvātajā uzvedības pamattipu sarakstā programmēšanas valodas iespēju salīdzināšanai (*Batdalov*, 2017; *Batdalov et al.*, 2016).

4.1. Vispārinātās funkcijas

Funkcija ir galvenā uzvedības vienība lielākajā daļā programmēšanas valodu. Funkcijas ļauj izstrādātājiem sadalīt skaitļošanu loģiski atsevišķās atkārtoti lietojamās daļās ar skaidri definētām ieejām un izvadēm. Papildus pamata bezstavokļa funkcijām ir arī paplašinājumi, kas dažādās valodās atšķiras. Funkcija vai funkcijai līdzīgs objekts var tikt saglabāts mainīgajā, nodots kā argumentu citām funkcijām (augstākas kārtas funkcijām) un saglabāt iekšējo stāvokli.

Galvenais piemērs ir parastā funkcija, kas pastāv praktiski jebkurā vispārējas nozīmes programmēšanas valodā. Valodās, kurās funkcijas tiek uzskatītas par "pirmās klases pilsoņiem", funkcijas (vai funkciju rādītāji) var glabāt mainīgajos, pārsūtīt starp kontekstiem un darīt visu, ko vien var darīt ar datiem. Dažas valodas atbalsta anonīmas funkcijas (lambda izteiksmes). Funktori (angl. *functors*), t. i., objekti, kuriem var būt iekšējais stāvoklis un kurus var "saukt" tā, it kā tie būtu funkcijas, aptver šablona vispārējo gadījumu. Funktorus var definēt programmas kodā vai automātiski ģenerēt kompilators (slēgumi (angl. *closures*) un ģeneratori (angl. *generators*)). Diemžēl dažās valodās dažādas funkcijas un funkcijām līdzīgi objekti nav pilnībā savstarpēji aizstājami, kas kavē pāreju starp tām.

Izmantotā formalizācija vispārināto funkciju definē kā universālo (ģenerisko) eksistenciālo (abstrakto) tipu, kas ļauj lietot objektu uz parametra tipa argumentu un saņemt atgriežamās vērtības tipa vērtību. Kā parasti tipu teorijā, vairāku parametru funkcijas netiek uzskatītas, jo tās ir viegli reducējamas līdz viena parametra funkcijām.

4.2. Izpildītāji

Izpildītājs (anlg. *Executor*) ir vides, kas var izpildīt kodu, abstrakcija, piemēram, pavediens, pavedienu pūls, testēšanas vide vai attālā sistēma. Izpildītājs nodrošina vienotu saskarni starpvides izsaukumiem. Šī saskarne nav ierobežota tikai ar parastajiem sinhroniem funkciju izsaukumiem; ir iespējami citi mijiedarbības stili (2.11. apakšnodaļa). Attālās sistēmas saskarne var būt ierobežota (piemēram, attālināti var izsaukt tikai noteiktas funkcijas). Vēl viena izpildītāju funkcija ir tāda, ka dažādiem izpildītājiem var būt dažādas globālo objektu (vienīgo) vērtības.

Standarta un specializētās bibliotēkās bieži ir saskarnes, lai atbalstītu pavedienus, pavedienu pūlus un attālo procedūru izsaukumus (angl. *remote procedure call*, *RPC*), taču reti ir kopīga saskarne visiem šiem izpildītājiem. Saskarnes definīcijas valodas definē ierobežotas saskarnes.

Globālo objektu (piemēram, viltus pulksteņu vai viltotu attālo sistēmu) aizstāšana ir izplatīta testēšanas vidēs.

Izmantotā formalizācija izpildītāju definē kā universālo (ģenerisko) eksistenciālo (abstrakto) tipu, kas ļauj izpildīt funkciju un izgūt (potenciāli modificējamu) izpildes kontekstu.

4.3. Datu piekļuves deleģēšana

Daudzu labi zināmu projektēšanas šablonu pamatā ir deleģēšana. Uzvedības deleģēšana parasti ir triviāla (tā vienkārši ir funkcija, kas izsauc citu funkciju), un tai nav nepieciešama atsevišķa apspriešana. Tomēr ir atsevišķs deleģēšanas gadījums – datu piekļuves deleģēšana. Dažas objektu īpašības var loģiski izskatīties kā datu atribūti, taču tās netiek saglabātas pašā objektā. Šajā gadījumā tas, vai īpašība tiek saglabāts, ir realizācijas detaļa, kas var mainīties, nemainot publisko saskarni. Lai paslēptu šīs detaļas, programmēšanas valoda var nodrošināt vienotu piekļuvi saglabātajiem un aprēķināmiem īpašībām.

Šādi aprēķināmas īpašības (angl. *computable properties*) tiek atbalstītas, piemēram, *C#*, *JavaScript*, *TypeScript*, *Scala* un *Python*. *Scala* papildus dokumentē konvenciju par blakusefektiem – metodi bez parametriem var deklarēt bez tukšām iekavām (`foo`, nevis `foo()`), kas nozīmē solījumu izvairīties no blakusefektiem.

Izmantotā formalizācija pieņem, ka katrai objekta īpašībai ir ieguvējs (angl. *getter*) un iestatītājs (angl. *setter*). Saglabātajām īpašībām tie ir triviāli, taču programmētājs var tos definēt sarežģītāk. Kompilatoram ir jāaizstāj piekļuve datiem kodā ar šo funkciju izsaukumiem.

4.4. Vienreiz pārejamais

Vienreiz pārejamā (anlg. *Traversable Once*) šablons ir Ēriha Gammas *et al.* aprakstītā iteratora (angl. *Iterator*) šablona (*Gamma et al.*, 1995) vispārinājums. Atšķirībā no iteratora šablona, tajā netiek pieņemts, ka iterējamās vērtības tiek saglabātas agregātā (krājumā). Tā vietā vērtības var izgūt vai aprēķināt pēc pieprasījuma. Šajā gadījumā iespēja šķērsot vienu un to pašu vērtību kopu otro reizi netiek garantēta. Tas ir iemesls, lai šablonu sauktu par vienreiz pārejamo. Ja pamatā esošā struktūra pieļauj vairākas pāriešanas, var izgūt vairākus vienreiz pārejamos objektus, katrs no kuriem tiek pāriets tikai vienu reizi. Vērtību izgūšana no vienreiz pārejamā var būt sinhrona vai asinhrona. Vienreiz pārejamais var lietot funkcionālu transformāciju (attēlojumu, angl. *mapping*) cita vienreiz pārejamā atgrieztām vērtībām.

Iterēšana caur konteineru (piemēram, masīvam, sarakstam vai kartei) ir viena no pamatdarbībām daudzās programmēšanas valodās. Iteratora šablona rašanās popularizēja *for* cikla atsevišķu formu, *for-each* ciklu, kas ir pielāgots iterēšanai. Dažas valodas (piemēram, *Python*, *C#* un *JavaScript*) atbalsta ģeneratorus, kur vērtībām nav jānāk no krājuma. *Scala* abas pieejas vispārina `TraversableOnce` tipā (par godu kuram tika izvēlēts šablona nosaukums). Asinhronais gadījums tiek atbalstīts retāk, taču *Python* un *JavaScript* ģeneratori var būt asinhroni. Vēl viens asinhrons piemērs ir `Observable` saskarne *ReactiveX* bibliotēkā.

Izmantotā formalizācija vienreiz pārejamo definē kā universālo (ģenerisko) eksistenciālo (abstrakto) tipu, kas ļauj no vērtības izveidot vienreiz pārejamo un izgūt vienu vērtību no esošā vienreiz pārejamā.

4.5. Vienreiz piešķiramais

Vienreiz piešķiramā (angl. *Assignable Once*) šablons tiek izmantots, lai pieprasītu vērtību, dodot vērtības radītājam ligzdu, kur vērtību var saglabāt. Tā ir vispārēja piešķiramās ligzdas abstrakcija, kas ir atsaistīta no konkrētā ligzdas (piemēram, mainīgā, konstantes vai nākotnes vērtības (angl. *future*)) rakstura. Tā kā ligzda pēc lietošanas var nebūt pieejama, šablonu sauc par vienreiz piešķiramo. Tāpat kā vienreiz pārejamā šablona gadījumā, ja lietojumprogrammas loģika pieļauj vairākas piešķires, vienreiz piešķiramos objektus var pieprasīt vairākas reizes. Vispārīgā gadījumā vienreiz piešķiramais ir pārnesams starp koda daļām un pārceļams laikā.

Programmēšanas valodās mainīgie ir izturīgas piešķiramās ligzdas. Šī šablona terminos katrai piešķires darbībai tie ģenerē vienreiz piešķiramo. Konstantes nodrošina tikai vienu vienreiz piešķiramo, kas ir jāizmanto inicializācijas laikā. Rādītāji un atsauces tipi atbalsta piešķiramo ligzdu pārsūtīšanu vai saglabāšanu vēlākai lietošanai (lai gan tie parasti neatbalsta vienreizējas piešķires garantijas). Solījumi (angl. *promises*) un nākotnes vērtības (angl. *futures*) atbalsta asinhrono gadījumu (parasti ar vienreizējas piešķires garantijām).

Izmantotā formalizācija vienreiz piešķiramo definē kā universālo (ģenerisko) eksistenciālo (abstrakto) tipu, kas atbalsta tikai vienu darbību – vērtības piešķiri. Nav vispārēja veida, kā izveidot piešķiramo ligzdu; tas ir atkarīgs no ligzdas rakstura un tādējādi tiek atlikts uz konkrētām realizācijām.

5. PIEŠĶIRE

Šajā nodaļā ir aprakstīti dažādu veidu piešķires šabloni. Piešķire ir viena no vissvarīgākajām darbībām programmēšanas valodās. Piešķire tiek saprasta plašā nozīmē – tā ietver mainīgā vai konstantes inicializēšanu, mainīgā vērtības atkārtotu piešķiri un objektu nodošanu starp kontekstiem (argumenta nodošanu funkcijai vai funkcijas atgriešanas vērtības atgriešanu). Visas šīs darbības maina saikni starp nosaukumiem un vērtībām programmā. Ir dažādi šīs saistības mainīšanas veidi (piemēram, vērtības kopēšana pret divu mainīgo liekšanu atsaukties uz vienu un to pašu atmiņas fragmentu), kas atbilst dažādiem šabloniem.

Identificētie šabloni balstīti autora piedāvātajā piešķires veidu sarakstā programmēšanas valodas pazīmju salīdzināšanai (*Batdalov*, 2017; *Batdalov et al.*, 2016), kas vēlāk tika aprakstīti šablonu veidā (*Batdalov and Nikiforova*, 2021).

5.1. Vērtības piešķire

Vērtības piešķire (angl. *Value Assignment*) ir visvienkāršākais piešķires veids – avota vērtība tiek tieši saglabāta mērķī. Prototipiskajā gadījumā programma kopē vērtību no avota uz mērķi. Ja vērtība ir saistītā datu struktūra, tas nozīmē datu struktūras atjaunošanu jaunos atmiņas gabalos (dziļā kopēšana, angl. *deep copying*). Šī darbība ļauj izvairīties no kopīgā stāvokļa, taču tā var būt dārga, un kompilatoram ir jāsaprot datu struktūras atmiņas iedalīšana. Dažas valodas kopē tikai struktūras bāzi, piemēram, koka saknes mezglu (seklā kopēšana, angl. *shallow copying*), kas padara šo darbību par vērtības un atsauces piešķires hibrīdu, kas nav aizsargāta pret kopīgā stāvokļa trūkumiem. Dažās situācijās seklā kopēšana ir droša, proti, nemaināmajiem tipiem vai tad, ja avots netiek izmantots pēc piešķires (avots tiek pārvietots uz mērķi).

Programmēšanas valodas primitīviem tipiem parasti izmanto vērtību piešķiri, taču situācija ar saliktajiem tipiem var būt sarežģītāka. *C++* izmanto vērtību piešķiri ar dziļo kopēšanu pat saliktajiem tipiem un izmanto pārvietošanas un atgriešanas vērtību optimizāciju, lai izvairītos no pārmērīgas kopēšanas. Tādas valodas kā *Java*, *JavaScript* un *C#* uzskata visus neprimitīvo tipu par atsauces tipiem un izmanto atsauces piešķiri. Ja šajās valodās ir nepieciešama vērtības piešķire atsauces tipam, programmētājam tā ir jārealizē manuāli. *Perl* atbalsta vērtību piešķiri saliktajiem tipiem, bet veic tikai seklo kopēšanu.

Izmantotā formalizācija reducē vērtības piešķiri līdz divām darbībām – vērtības tipa piešķiramās ligzdas iegūšana mērķī un ligzdas izmantošana avota vērtības piešķirei.

5.2. Atsauces piešķire

Atsauces piešķire (angl. *Referential Assignment*) nozīmē, ka mērķa simbols (nosaukums) sāk atsaukties uz avota atmiņas vietu. Rezultātā abi simboli norāda uz vienu un to pašu atmiņas vietu. Tas bieži ir efektīvāks par vērtību piešķiri un atbalsta nekopējamu datu (piemēram, resursu turus (angl. *handles*)) atkārtotu izmantošanu dažādās programmu daļās. Tomēr atsauces piešķire var radīt kopīgo maināmo stāvokli, izraisot tādas nelabvēlīgas sekas kā sacīkšu apstākļi

(angl. *race conditions*) daudzpavedienu vidē, funkcijai nodota argumenta nejauša pārveidošana un atmiņas gabala izmantošana pēc tās atbrīvošanas. Sarežģītas atmiņas pārvaldības sistēmas, piemēram, atkritumu savākšanas (angl. *garbage collection*) vai viedo rādītāju (angl. *smart pointers*), prombūtnē atsauces piešķire var izraisīt atmiņas noplūdes (angl. *memory leaks*) vai karājošās atsauces (angl. *dangling references*).

C++ valodā piešķires operators nozīmē vērtību piešķiri, bet rādītāju un atsauču piešķire realizē atsauces piešķiri. Tādas valodas kā *Java*, *JavaScript* un *C#* neprimitīvus tipus vienmēr uzskata par atsaucēm, tāpēc to piešķire ir atsauces piešķire.

Atsevišķa atsauces piešķires formalizācija ir lieka, jo tehniski tā ir vienkārši atsauces vērtības piešķire. Pietiek, lai tipu sistēma atbalstītu atsauces tipus. Šī pieeja precīzi atspoguļo to, kā atsauces piešķire parasti ir **realizēta** programmēšanas valodās. Aprakstītais šablons nodrošina atšķirīgu **skatienu** uz šo mehānismu, bet tā realizācija un formalizācija nemainās.

5.3. Daļējā piešķire

Daļējā piešķire (angl. *Partial Assignment*) rodas, ja ir jāmaina saliktās datu struktūras (piemēram, masīva, ieraksta vai kartes) daļa. Maināmu datu struktūru var modificēt vietā, un nemaināmas struktūras izmanto kopēšanas ar maiņu (angl. *copy-change*) darbību (jauna objekta, kur piešķirtajai daļai ir jaunā vērtība un citām daļām ir vecās vērtības, izveide). Pēdējā gadījumā visas nemainītās daļas kopēšana var būt dārga, tāpēc izveidotā struktūra parasti vismaz daļēji koplieto datus ar veco struktūru (jo nemaināmajiem tipiem kopīgs stāvoklis ir drošs).

Programmēšanas valodās daļējā piešķiršana ir raksturīga konteineriem un konteineriem līdzīgiem objektiem, kas uztur kādu atslēgas un vērtības vai indeksa un vērtības asociācijas veidu (piemēram, masīviem, sarakstiem, ierakstiem un kartēm). Šo tipu nemaināmie ekvivalenti parasti atbalsta kopēšanas ar maiņu darbību. Daļējās modifikācijas līdzsvarošana ar izvairīšanos no pārmērīgas kopēšanas var būt sarežģīta. Piemēram, `Vector` klase, nemaināms masīva līdzinieks *Scala* standarta bibliotēkā, ir iekšēji realizēts kā koks ar zarošanās koeficientu 32 (*Odersky and Spoon*, 2023).

Izmantotā formalizācija pieņem, ka saliktais tips ģenerē piešķiramu ligzdu modificējamam elementam (vai nu iterācijas laikā, vai ar indeksu vai atslēgu), un tā tiek izmantota piešķirei.

5.4. Destrukturēšana

Destrukturēšana (angl. *Destructuring*) atbalsta saliktās datu struktūras sadalīšanu un tās elementu piešķiri vairākiem mērķiem vienā priekšrakstā. Tā ir "sintaktiskais cukurs" (angl. *syntactic sugar*) un nepievieno jaunu funkcionalitāti. Tomēr tā var vienkāršot kodu, it īpaši, ja avota datu struktūra nāk no citas funkcijas izsaukuma. Piešķire atsevišķiem mērķiem var būt vērtība vai atsauces piešķire. Dažas avota daļas var nebūt jāuzglabā, tāpēc destrukturēšana tās var izlaist. Lietotāja definēto tipu destrukturēšanas piešķire var prasīt šo tipu veidotāju papildu realizācijas darbu (ja valoda to atbalsta).

Standarta tipu (piemēram, masīvu) destrukturēšanas piešķire tiek atbalstīta, piemēram, *C++*, *JavaScript*, *TypeScript* un *Python*. *Scala* un *Kotlin* atbalsta lietotāja definētu tipu destrukturēšanu, ja tie ietver atbilstošas destrukturēšanas metodes.

Izmantotā formalizācija pieņem, ka katrai salikto objektu no vairākām vērtībām izveidojošai darbībai, ir līdzinieks pretējai darbībai. Pēc tam kompilators var izsaukt šo ekvivalentu un piešķirt atgrieztās vērtības mērķa mainīgajiem.

5.5. Izsaiņošana

Izsaiņošana (angl. *Unboxing*) vienkāršo sintaksi, izmantojot citā objektā glabātu vērtību, piemēram, lietojumprogrammas saskarnes izsaukuma rezultātu solījumā (angl. *promise*) vai nākotnes vērtībā (angl. *future*). Līdzīgi destrukturēšanai, izsaiņošanas piešķire ir tikai "sintaktiskais cukurs", bet padara programmas sintaksi kodolīgāku. Atšķirībā no citiem piešķires operatoriem, izsaiņošanas piešķire nav parasta funkcija. Tā vietā, saskaroties ar izsaiņošanas piešķiri, kompilators maina izpildes plūsmu. Piemēram, izsaiņojot vērtību no solījuma, kompilators pārvērš turpmāko funkcijas kodu lambda izteiksmē, kas jāizpilda pēc tam, kad solījums ir saņēmis vērtību.

Gaidīšanas operators (`await`) *C#*, *JavaScript*, *TypeScript* un *Python* izsaiņo nākotnes vērtības tipu (`Task`, `Promise` vai `Future`). Tas atbalsta arī lietotāja definētus tipus, kas nosaka atbilstošās metodes. *Scala* izmanto vispārīgāku izsaiņošanas gadījumu ar *for* izpratnēm (angl. *for-comprehensions*), ko var izmantot ar patvaļīgiem `map`, `flatMap` un `filter` metodes definējošiem tipiem (piemēram, visām standarta kolekcijām).

Izmantotā formalizācija apraksta transformāciju, kas jāveic kompilatoram, kas definēta kā *Scala* gadījumā. Šīs transformācijas precīzā semantika ir atkarīga no saiņotā tipa.

6. PĀRBAUDE

Promocijas darbs ilustrē aprakstītās tipu sistēmas iespējas, izmantojot *MIX*, mītiska datora, ko Donalds Knuts izgudroja savai grāmatu sērijai "Datorprogrammēšanas māksla" (*Knuth*, 1997), emulatora piemēru. Neskatoties uz to, ka tas ir "rotaļlietas" projekts, tas parāda iespējamos uzlabojumus pat tādā izteiksmīgā valodā kā *Scala* (*Batdalov and Nikiforova*, 2017). Paredzamās piedāvāto uzlabojumu lielāka izteiksmīguma praktiskās sekas ir realizācijas sarežģītības samazināšana. Apspriešana zināmā mērā ir spekulatīva, jo promocijas darbā ir aprakstīta tikai tipu sistēma, nevis faktiskā programmēšanas valoda, un aplūkotās koda izmaiņas atšķiras no sistēmas faktiskās vēstures. Neskatoties uz to, šī diskusija parāda iespējamo palīdzību, ko varētu sniegt aprakstītā tipu sistēma.

6.1. Projekta apraksts

Donalds Knuts izgudroja *MIX* iedomāto datoru savai grāmatu sērijai "Datorprogrammēšanas māksla" (*Knuth*, 1997). Lielākā daļa sērijas koda fragmentu un vingrinājumu risinājumu ir rakstīti *MIX* asemblera valodā (angl. MIX assembler language, *MIXAL*).

Būtiska *MIX* iezīme ir tā nepilnais determinisms. *MIX* var darboties kā binārs vai decimāls dators, un pareizai programmai jābūt neatkarīgai no baita lieluma (*Knuth*, 1997). Tāpat programmām ir jādarbojas pareizi neatkarīgi no asinhrono ievadizvades darbību ātruma. Šis indeterminisms rada grūtības programmēšanā zema līmeņa asemblera valodā.

Autors izstrādāja tīmekļa *MIX* emulatoru ar papildu pareizības pārbaudes funkcijām (*Batdalov and Nikiforova*, 2017). Emulators var izpildīt programmas binārajā vai decimālajā režīmā un pārbaudīt ievadizvades sinhronizācijas pareizību. Tas var arī ierakstīt katru stāvokli programmas izpildlaikā un pārslēgties starp tiem uz priekšu un atpakaļ (šo funkciju iedvesmojis *Online Python Tutor* (*Guo*, 2013)). Šie līdzekļi nodrošina rīkus, lai pārbaudītu, vai programma atbilst *MIX* arhitektūras noteiktajiem noteikumiem.

Lai atbalstītu baita lieluma mainīgumu, emulators definē dažādas no baita lieluma atkarīgo klašu ģimenes (`MixByte`, `MixWord` un citas) binārajam un decimālajam režīmam un izmanto *Scala* valodas ģimenes polimorfismu (angl. *family polymorphism*) (*Odersky et al.*, 2006). Pēdējais ir nepieciešams, lai nodrošinātu to, ka vienas ģimenes klases tiek izmantotas kopā (piemēram, binārais reģistru stāvoklis nav saderīgs ar decimālajiem vārdiem). Lietotājs var palaist emulatoru binārajā vai decimālajā režīmā un pārbaudīt, vai rezultāts ir vienāds.

Lai pārbaudītu ievadizvades sinhronizācijas pareizību, emulators izmanto atmiņas bloķēšanu, ko iedvesmo *SQL* datu bloķēšana (*ISO/IEC 9075-2:2023*, 2023). Kad programma uzsāk ievadizvades darbību, emulators bloķē atbilstošo atmiņas apgabalu un neļauj veikt darbības, kuru rezultāti ir atkarīgi no tā, vai ievadizvades darbība ir pabeigta.

Emulatora stāvokļa izsekošana un atgriešanās iepriekšējos stāvokļos tiek panākta, saglabājot katru stāvokli nemaināmajās datu struktūrās. Lai programmas izpildlaikā ģenerētu

jaunu stāvokli, tiek lietota kopēšanas ar maiņu darbība. Tā kā nemaināmās datu struktūras izmanto kopīgo atmiņu, visu stāvokļu saglabāšana nav ļoti dārga.

Autora realizācijas pirmkods (*Scala* un *TypeScript* valodās) ir pieejams vietnē *https://github.com/linnando/MIXEmulator*. Emulatora darbojošās kopija ir pieejama vietnē *https://www.mix-emulator.org*.

6.2. Koda vienkāršošana

Iespējamais vienkāršojums, ko piedāvātā tipu sistēma varētu nodrošināt, ir saistīts ar maināmo un nemaināmo strukturālo tipu (aprakstīts 3. nodaļā, kā arī to pārmantotāju) realizāciju iespēju paritāti. Cita starpā, iespēju paritāte nozīmē, ka gan maināmie, gan nemaināmie tipi atbalsta daļējo piešķiri (piešķiri saliktā objekta daļai; 5.3. apakšnodaļa).

Daļējā piešķire tiek plaši izmantota emulatora kodā, jo stāvoklis pēc katras darbības ir iepriekšējā stāvokļa modifikācija. Tomēr nemaināmo tipu daļējai maiņai ir nepieciešami sarežģīti priekšraksti, piemēram, `copy(forwardReferences = forwardReferences.updated(symbol, forwardReferences(symbol) :+ counter))`. Iemesls ir tāds, ka nemaināmā objekta kopēšanas ar maiņu darbība nav reducējama līdz viena elementa maiņai, kā tas ir maināmajā gadījumā. Tā vietā tā ietver visas datu struktūras maiņu, sākot no saknes.

5.3. apakšnodaļā aprakstītā pieeja piedāvā tiešu atbalstu nemaināmo datu tipu daļējai piešķirei. Tas ietver piešķiramās ligzdas ģenerēšanu, kurā būtu rādītājs uz visu datu struktūru, nevis piešķiramo mezglu (kopā ar informāciju par mezgla atrašanu). Vērtības piešķire šai ligzdai varētu izmainīt datu struktūru un saglabāt atsauci mērķī. Tad kods, kas modificē emulatora stāvokli, varētu izskatīties tikpat vienkāršs kā maināmajā gadījumā: `forwardReferences(symbol) :+= counter`.

6.3. Attīstība no vienkāršāka prototipa

Tā kā viens no piedāvātās tipu sistēmas mērķiem ir atbalstīt koda attīstību (un ne tikai vienkāršot vienreiz uzrakstīto kodu), ir svarīgi izprast iespējamos vienkāršojumus emulatora izstrādē no vienkāršota dizaina līdz pašreizējam stāvoklim. Šis domu eksperiments nav tīrs, jo tas neatbilst faktiskajai emulatora vēsturei, taču tam ir jēga, jo šis izstrādes veids ir raksturīgs lietojumprogrammām.

Ja emulators sākotnēji būtu bijis tikai binārs un decimālais režīms būtu pievienots vēlāk, būtu jāpievieno jaunas no baitu lieluma atkarīgo klašu realizācijas. Izmantojot tradicionālo objektorientēto polimorfismu, jaunu realizāciju pievienošana būtu bijusi vienkārša, taču emulators izmanto *Scala* ģimenes polimorfismu. Iemesls ir tāds, ka dažādu ģimeņu klases nav savietojamas un nevar būt viena un tā paša abstrakta tipa realizācijas. Lai tos padarītu par viena un tā paša abstraktā tipa realizācijām, būtu nepieciešami kovariantie metožu argumenti, kas nav tipu droši.

Tomēr *Scala* ģimenes polimorfisms prasa, lai visa savstarpēji saistītu klašu ģimene (piemēram, visa binārā realizācija) būtu ietverta vienīgajā objektā. Netriviālas loģikas gadījumā

tas rada milzīgus objektus, pārkāpjot vienotās atbildības principu (*Batdalov and Nikiforova*, 2017). Tādējādi ģimenes polimorfisma realizācija esošā sistēmā būtu sarežģīta.

Promocijas darbā ir piedāvāta alternatīva, kas saistīta ar pārmantošanas interpretāciju lietošanas vietas variantuma terminos, kas aplūkota 2.7. apakšnodaļā. Šī pieeja būtībā atļāva metožu parametru apakštipēšanu dziļuma apakštipēšanas gadījumā ar nosacījumu, ka attiecības starp parametriem un atgriešanas vērtību tipiem garantē tipu drošumu. To pašu pieeju var lietot ģimenes polimorfismam. Pēc tam saskarni un vairākas implementācijas var definēt saistītās, bet atsevišķās klasēs, padarot jaunas realizācijas pievienošanu tikpat ērtu kā tipiskajā objektorientētā gadījumā.

Vēl viens potenciāls vienkāršojums ir rīkošanās ar vienīgajiem (angl. *singletons*) un izpildītājiem (angl. *executors*). Pašreizējā realizācija nodod apstrādes modeli (bināro vai decimālo) kā parametru to izmantojošām funkcijām. Tikai binārā emulatorā apstrādes modeļa elementi, šķiet, būtu bijuši globāli pieejami kā atsevišķie elementi. Lai pārietu no šāda projektējuma uz projektējumu ar vairākiem apstrādes modeļiem, būtu nepieciešama ievērojama pārstrukturēšana dažādās koda daļās.

Kā alternatīva 4.2. apakšnodaļā ir aprakstīti izpildītāji, kuri, cita starpā, satur vienīgos. Vienīgie ir sasniedzami kā globāli objekti, taču programma tos var aizstāt izpildlaikā. Pēc tam kods, kas izvēlas emulatora režīmu, varētu iestatīt apstrādes modeli izpildītājā, un pārējais kods tam varētu piekļūt tāpat kā iepriekš.

6.4. Iespējamā nākotnes attīstība

Iespējamais emulatora uzlabojums nākotnē ir saistīts ar tā ievadizvades darbību noteiktības interpretāciju. Emulatora izvirzītie nosacījumi ir ļoti ierobežojoši. Piemēram, tas uzskatītu programmu, kas periodiski ieraksta `WAITING...` terminālī līdz ievadizvades darbības pabeigšanas, par nedeterministisku.

Emulators varētu būt pieļaujošāks, ja tas varētu izsekot iespējamo virtuālās mašīnas stāvokļu diapazonam, nevis precīzam stāvoklim. Kamēr iespējamo stāvokļu atšķirība saglabājas saprātīga, izpilde turpinās (kas tieši ir saprātīgs, ir izvēles jautājums). Tāda pati pieeja būtu noderīga baita lieluma nenoteiktības gadījumā. Tas novērstu prasību atsevišķi palaist programmu binārajā un decimālajā režīmā.

Tomēr izsekošana stāvokļu diapazonam vairumā gadījumu ir sarežģīta un nepamatota, jo lielākā daļa programmu ir deterministiskas. Emulators varētu izsekot precīzam stāvoklim, cik ilgi vien iespējams, un pārslēgties uz stāvokļa robežu režīmu, kad darbība vairs nav deterministiska. Šie divi režīmi ievērojami atšķirtos, tāpēc ir lietderīgi izmantot stāvokļa (angl. *State*) šablonu (*Gamma et al.*, 1995). 2.9. apakšnodaļā aprakstīti hameleonu objekti, kas atvieglotu tā realizāciju. Virtuālās mašīnas deterministiskās un nedeterminiskās realizācijas būtu dažādi stāvokļi, kurus vajadzības gadījumā var pārveidot savā starpā.

NOBEIGUMS

Promocijas darbs ir veltīts saiknes noteikšanai starp projektēšanas šabloniem un programmēšanas valodu izteiksmes iespējām. Kā zināms no literatūras un pieredzes, šablonos balstīti projektējumi bieži ir saistīti ar lielāku sarežģītību, lai gan to sākotnējais mērķis bija pretējs. Izvirzītā hipotēze ir tāda, ka pārmērīgo sarežģītību daļēji izraisa programmēšanas valodu nepietiekams izteiksmīgums. Valodām ir grūtības šablonu attēloto domu konstrukciju atveidošanā. Lai risinātu šo problēmu, promocijas darbā ir piedāvāts dažādu programmēšanas valodu konstrukciju vispārīgos gadījumus aprakstošu vispārinājumu kopums un to tipu teorijas formālismus izmantojoša formalizēšana.

Šim darbam noteiktie uzdevumi ir pilnībā izpildīti.

1. Programmēšanas valodu attīstības tendenču un iepriekš aprakstīto grūtību projektēšanas šablonu realizācija analīze atklāja problemātiskus punktus pat mūsdienu programmēšanas valodās.
2. Pamatojoties uz šo analīzi, tika formulēti mērķi un prasības vēlamajai tipu sistēmai.
3. Tika aprakstīti datu salikšanas un pamata skaitļošanas primitīvu šabloni, kas sniedz biežāk lietoto programmēšanas valodu konstrukciju vispārinājumus.
4. Aprakstītie modeļi tika formalizēti, izmantojot tipu teorijas aparātu.
5. *MIX* datora emulators ir realizēts *Scala* valodā, lai to izmantotu kā praktisku projekta piemēru.
6. Tiek parādīts, kā piedāvātie tipi varētu būt noderīgi, realizējot piemērprojektu un tā attīstībā.

Promocijas darba galvenais rezultāts ir izstrādāta tipu sistēma, kas reprezentē pamata strukturālās un skaitļošanas šablonus. Šabloni apraksta programmēšanas valodām tipisku konstrukciju vispārīgus gadījumus (lai gan faktiskās valodas pašlaik var atbalstīt šablonu pilnībā vai tikai tā atsevišķus gadījumus) un tādējādi nodrošina elastību programmētāju domu modelēšanā. Šādu primitīvu esamība reālās programmēšanas valodās varētu veicināt projektēšanas šablonu realizāciju un, vēl svarīgāk, esošas sistēmas tālāko attīstību.

Darba papildu rezultāti

1. Aprakstīto konstrukciju apraksti šablonu veidā izskaidro, kā un kāpēc šīs konstrukcijas tiek lietotas.
2. Zināmo lietojumu saraksts salīdzina un kontrastē aprakstīto šablonu realizācijas dažādās valodās. Tas atvieglo līdzvērtīgu vai līdzīgu konstrukciju atrašanu dažādās valodās un to ierobežojumu izpratni.
3. Valodu vai bibliotēku primitīvu kā šablonu aprakstīšanas metodoloģiju, lai aptvertu vispārīgus gadījumus un pēc tam tos formalizētu kā tipus, var izmantot arī citām konstrukcijām, kas nav aplūkotas promocijas darbā.
4. Izstrādāto *MIX* emulatoru var lietot mācībās. Tam ir noteiktas priekšrocības, salīdzinot ar citiem emulatoriem, piemēram, decimālā režīma atbalsts (papildus binārajam) un ievadizvades sinhronizācijas pareizības pārbaude.

Pamatojoties uz veikto pētījumu, var izdarīt vairākus secinājumus. Tie ir:

1. Programmēšanas valodu konstrukcijas bieži atspoguļo realizēto vispārīgo šablonu tikai atsevišķus gadījumus. Tomēr izteiksmīgākas valodas var atbalstīt šos šablonus pilnā veidā.
2. Valodas konstrukciju vispārīgākus gadījumus var identificēt, aprakstot tās kā šablonus.
3. Aprakstītie šabloni der tipu teorijas formalizācijai, kas potenciāli ļauj tiem kalpot kā valodas konstrukcijas.
4. Pat izteiksmīgākajās programmēšanas valodās, piemēram, *Scala*, ir vieta uzlabojumiem attiecībā uz aprakstītajiem šabloniem.

Darbu var turpināt vairākos virzienos. Tie ir:

1. To pašu metodiku var lietot arī citām konstrukcijām. Piemēram, promocijas darbs neaptver tādus būtiskus programmēšanas valodas aspektus kā objekta dzīves cikls un atmiņas pārvaldība.
2. Promocijas darbā aprakstītos vispārīgos šablonus var ieviest jaunās un esošās programmēšanas valodās.

BIBLIOGRĀFIJA

Moez A. AbdelGawad. A comparison of NOOP to structural domain-theoretic models of object-oriented programming. Preprint available at https://arxiv.org/abs/1603.08648, 2017.

Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Angel Shlomo. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press USA, 1977. ISBN 978-0-19-501919-3.

Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ in-depth series. Addison-Wesley, 2001. ISBN 978-0-201-70431-0.

Pavol Bača and Valentino Vranić. Replacing object-oriented design patterns with intrinsic aspect-oriented design patterns. In *Proceedings of the 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC)*, pages 19–26. IEEE, 2011. doi: 10.1109/ecbs-eerc.2011.13.

Ruslan Batdalov. Inheritance and class structure. In Pavel P. Oleynik, editor, *Proceedings of the First International Scientific-Practical Conference Object Systems – 2010*, pages 92–95, 2010. URL https://cyberleninka.ru/article/n/inheritance-and-class-structure/pdf.

Ruslan Batdalov. Is there a need for a programming language adapted for implementation of design patterns? In *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP '16)*, pages 34:1–34:3. Association for Computing Machinery, 2016. ISBN 978-1-4503-4074-8. doi: 10.1145/3011784.3011822.

Ruslan Batdalov. Comparative analysis of object-oriented programming languages in the context of language expressiveness. Master's thesis, Riga Technical University, 2017.

Ruslan Batdalov and Oksana Nikiforova. Towards easier implementation of design patterns. In *Proceedings of the Eleventh International Conference on Software Engineering Advances (ICSEA 2016)*, pages 123–128. IARIA, 2016.

Ruslan Batdalov and Oksana Ņikiforova. Implementation of a MIX emulator: A case study of the Scala programming language facilities. *Applied Computer Systems*, 22(1):47–53, 2017. doi: 10.1515/acss-2017-0017.

Ruslan Batdalov and Oksana Nikiforova. Three patterns of data type composition in programming languages. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18)*, pages 32:1–32:8. Association for Computing Machinery, 2018. ISBN 978-1-4503-6387-7. doi: 10.1145/3282308.3282341.

Ruslan Batdalov and Oksana Nikiforova. Elementary structural data composition patterns. In *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP '19)*, pages 26:1–26:13. Association for Computing Machinery, 2019. ISBN 978-1-4503-6206-1. doi: 10.1145/3361149.3361175.

Ruslan Batdalov and Oksana Nikiforova. Patterns for assignment and passing objects between contexts in programming languages. In *Proceedings of the 26th European Conference on Pattern Languages of Programs (EuroPLoP '21)*, pages 4:1–4:9. Association for Computing Machinery, 2021. ISBN 978-1-4503-8997-6. doi: 10.1145/3489449.3489975.

Ruslan Batdalov, Oksana Nikiforova, and Adrian Giurca. Extensible model for comparison of expressiveness of object-oriented programming languages. *Applied Computer Systems*, 20(1):27–35, 2016. doi: 10.1515/acss-2016-0012.

Judith Bishop. *C# 3.0 Design Patterns*. O'Reilly Media, Sebastopol, CA, USA, 2008. ISBN 978-0-596-52773-0.

Grady Booch. The well-tempered architecture. *IEEE Software*, 24(4):24–25, 2007. ISSN 0740-7459. doi: 10.1109/ms.2007.122.

Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998. ISSN 0896-8438.

Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing*, volume 4 of *Pattern-Oriented Software Architecture*. Wiley, 2007a. ISBN 978-0-470-06530-3.

Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture, On Patterns and Pattern Languages*, volume 5 of *Pattern-Oriented Software Architecture*. Wiley, 2007b. ISBN 978-0-470-51257-9.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, A System of Patterns*, volume 1 of *Pattern-Oriented Software Architecture*. Wiley, 2013. ISBN 978-1-118-72526-9.

Travis Carlson and Eric Van Wyk. Type qualifiers as composable language extensions for code analysis and generation. *Journal of Computer Languages*, 50:49–69, 2019. ISSN 2590-1184. doi: 10.1016/j.jvlc.2018.10.008.

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, 1989.

James O. Coplien. *Software Patterns*. SIGS management briefings. SIGS, 1996. ISBN 978-1-884842-50-4.

Ivica Crnković, Severine Séntilles, Aneta Vulgarakis, and Michel R. V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, 2011. ISSN 0098-5589. doi: 10.1109/tse.2010.83.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320.

ECMA-334. *C# Language Specification*. Ecma International, sixth edition, 2022. Standard.

Francesca Arcelli Fontana, Stefano Maggioni, and Claudia Raibulet. Understanding the relevance of micro-structures for design patterns detection. *Journal of Systems and Software*, 84(12):2334–2347, 2011. ISSN 0164-1212. doi: 10.1016/j.jss.2011.07.006.

Francesca Arcelli Fontana, Stefano Maggioni, and Claudia Raibulet. Design patterns: a survey on their micro-structures. *Journal of Software-Evolution and Process*, 25(1):27–52, 2013. ISSN 2047-7481. doi: 10.1002/smr.547.

Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2012. ISBN 978-0-13-306521-3.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995. ISBN 978-0-201-63361-0.

Joseph Gil and David H. Lorenz. Design patterns and language design. *Computer*, 31(3):118–120, 1998. ISSN 0018-9162. doi: 10.1109/2.660196.

The Go Programming Language Specification. Google, 2023.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification. Java SE 8 Edition*, 2015.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification. Java SE 21 Edition*, 2023.

Philip J. Guo. Online Python Tutor: Embeddable web-based program visualization for CS education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1868-6. doi: 10.1145/2445196.2445368.

Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. *ACM Sigplan Notices*, 37(11):161–173, 2002. ISSN 0362-1340. doi: 10.1145/583854.582436.

Tony Hoare. Null references: The billion dollar mistake. Available at https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare, August 2009.

ISO/IEC 14882:2020. *Information technology – Programming languages – C++*. ISO/IEC, 2020. Standard.

ISO/IEC 9075-2:2023. *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*. ISO/IEC, 2023. Standard.

Kotlin Language Documentation 1.9.0. JetBrains, 2023. URL https://kotlinlang.org/docs/kotlin-reference.pdf.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the European conference on object-oriented languages (ECOOP '97)*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69127-3.

Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture, Patterns for Resource Management*, volume 3 of *Pattern-Oriented Software Architecture*. Wiley, 2013. ISBN 978-1-118-72523-8.

Donald E. Knuth. *The Art of Computer Programming: Fundamental algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley, 1997. ISBN 978-0-201-89683-1.

Chris Lattner and Vikram S. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO 2004)*, pages 75–86, New York, NY, USA, 2004. IEEE. doi:10.1109/CGO.2004.1281665.

António Leitão and Sara Proença. On the expressive power of programming languages for generative design: the case of higher-order functions. In *Proceedings of the 32nd International Conference on Education and Research in Computer Aided Architectural Design in Europe (eCAADe)*, pages 257–266, 2014.

George Leontiev, Eugene Burmako, Jason Zaugg, Adriaan Moors, Paul Phillips, Oron Port, and Miles Sabin. SIP-23 – literal-based singletone types. Scala Improvement Proposal, 2019.

Fredrik Skeel Løkke. Scala & design patterns. Master's thesis, University of Aarhus, March 2009.

Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Pearson Education, 2008. ISBN 978-0-13-608325-2.

Greg Michaelson. *An introduction to functional programming through lambda calculus*. Dover Publications, Inc., 2011. ISBN 978-0-486-47883-8.

TypeScript Documentation. Microsoft Corporation, 2023. URL https://www.typescriptlang.org/docs/.

Miguel P. Monteiro and João Gomes. Implementing design patterns in Object Teams. *Software: Practice and Experience*, 43(12):1519–1551, 2013. ISSN 1097-024X. doi:10.1002/spe.2154.

Peter D. Mosses. Formal semantics of programming languages: An overview. *Electronic Notes in Theoretical Computer Science*, 148(1):41–73, 2006. ISSN 1571-0661. doi:10.1016/j.entcs.2005.12.012.

Martin Odersky and Lex Spoon. *Scala Collections*, 2023. URL http://docs.scala-lang.org/overviews/collections/introduction.html.

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the Scala programming language. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

Martin Odersky, Philippe Altherr, Vincent Cremet, Sébastien Doeraene, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. Scala Language Specification: Version 3.4, 2023. URL https://scala-lang.org/files/archive/spec/3.4/.

Unified Modeling Language Version 2.5.1. OMG, 2017. URL https://www.omg.org/spec/UML/2.5.1. Standard.

Java® Platform, Standard Edition Core Libraries Release 21. Oracle, 2023. URL https://docs.oracle.com/en/java/javase/21/core/java-core-libraries1.html.

PHP 8.2 Language Reference. PHP Group, 2023.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design And Implementation*. Prentice Hall PTR, fourth edition, 2001. ISBN 978-0-13-027678-0.

The Python Language Reference 3.12.0. Python Software Foundation, 2023.

Dimitri Racordon and Didier Buchs. Implementing a language with explicit assignment semantics. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2019)*, page 12–21, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6987-9. doi:10.1145/3358504.3361227.

Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2 of *Pattern-Oriented Software Architecture*. Wiley, 2013. ISBN 978-1-118-72517-7.

Peter Sommerlad. Design patterns are bad for software design. *IEEE Software*, 24(4):68–71, 2007. ISSN 0740-7459. doi: 10.1109/ms.2007.116.

Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, sep 1996. ISSN 0360-0300. doi: 10.1145/243439.243441.

Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992a.

Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. Association for Computing Machinery, 1992b.

Uwe Zdun and Paris Avgeriou. A catalog of architectural primitives for modeling architectural patterns. *Information and Software Technology*, 50(9-10):1003–1034, 2008. ISSN 0950-5849. doi: 10.1016/j.infsof.2007.09.003.

Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. Interface representation patterns: Crafting and consuming message-based remote APIs. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17)*, pages 27:1–27:36. Association for Computing Machinery, 2017. ISBN 978-1-4503-4848-5. doi:10.1145/3147704.3147734.

Ruslans Batdalovs (Ruslan Batdalov) dzimis 1983. gadā Iževskā (Krievija). Kazaņas Valsts universitātē ieguvis matemātiķa un sistēmu programmētāja kvalifikāciju lietišķajā matemātikā un informatikā (2003; ar izcilību), Rīgas Tehniskajā universitātē (RTU) – maģistra grādu datorsistēmās (2017; ar izcilību). Strādājis SIA "Gudrā māja", IT uzņēmumā "*CBOSS*", Maskavas pilsētas tālruņu tīklā, RTU un AS "Rietumu Banka". Patlaban ir *Google* programmatūras inženieris. Kopš 2010. gada ir *ACM* loceklis. Zinātniskās intereses saistītas ar projektēšanas šabloniem un programmēšanas valodu izteiksmību.