Универзитет у Београду
Математички факултет

Саша Малков

Повезивање строго типизираних функционалних програмских језика и релационих база података

Докторска дисертација

Београд
2009.

Универзитет у Београду – Математички факултет

Докторска дисертација

Аутор: Саша Малков

Наслов дисертације: Повезивање строго типизираних функционалних програмских језика и релационих база података

Ментор: др Ненад Митић,
Универзитет у Београду – Математички факултет

Чланови комисије: др Гордана Павловић-Лажетић
Универзитет у Београду – Математички факултет

др Мирјана Ивановић
Универзитет у Новом Саду – Природно-математички факултет

Области: програмски језици, функционално програмирање, базе података

Кључне речи: функционално програмирање, програмски језици, типови, строга типизираност, проверавање типова, аутоматско извођење типова, базе података

Датум: 15. мај 2009.

Резиме

Тема дисертације је разматрање теоријских и практичних аспеката повезивања строго типизираних функционалних програмских језика и релационих база података. Главне области обухваћене радом су функционални програмски језици, релационе базе података и типови.

При изради дисертације је спроведено истраживање чији је предмет била анализа аспеката повезивања и могућности имплементације. Тема дисертације је усмерена према строго типизираним функционалним програмским језицима, али у многим сегментима истраживања су разматрани и општи аспекти повезивања програмских језика и база података. Једна од основних хипотеза је била да се повезивање може заснивати на очувању међусобне ортогоналности програмског и упитног језика, тако да се повезивањем не доведу у питање њихове основне особине.

У раду су представљене значајне карактеристике функционалних програмских језика и релационих база података. Такође, представљени су алгоритми за аутоматско извођење типова. Описане су основне карактеристике програмског језика *Wafl*, који је послужио као платформа за истраживање и експериментисање, као и њему прилагођен алгоритам за извођење типова. Анализирани су и продискутовани различити аспекти повезивања програмских и упитних језика и изведени принципи и концепти на којима је потребно да почива њихово *добро* повезивање. Као основни принципи, на којима је потребно да се заснива повезивање, препознати су апстрактност, ортогоналност и симетричност. Препознати су и концепти путем којих се повезивање може остварити уз пуну сагласност са наведеним принципима: употреба упитног језика у програмском језику, употреба програмског језика у упитном језику, строга типизираност повезивања, јединственост трансакционог простора, ортогоналност у односу на типове података и ортогоналност у односу на начине повезивања.

Сваки од концепата је разрађен и експериментално потврђен имплементирањем у програмском језику *Wafl*. У раду су представљене синтаксне конструкције којима су допуњени програмски језик *Wafl* и упитни језик *SQL* ради имплементирања представљених концепата, као и њихова семантика. Обликован је алгоритам за статичко проверавање типова у програмима који користе базу података, у чијој имплементацији сарађују преводилац за програмски језик и систем за управљање базама података.

Међу најважније доприносе рада спадају препознавање принципа и концепата доброг повезивања програмских и упитних језика, потпуно нов концепт матичних

израза који се израчунавају на страни СУБП-а, имплементација записивања података сложених типова у бази података и одговарајуће пресликавање типова, концепт и имплементација јединственог трансакционог простора и представљени алгоритми за статичко извођење типова. Резултати овог истраживања могу да нађу примену како у даљем истраживању тако и непосредно у практичној употреби повезивања програмских и упитних језика.

Кључне речи: функционално програмирање, релационе базе података, типови, строга типизираност, проверавање типова, аутоматско извођење типова, програмски језици

Abstract

The theme of the dissertation is consideration of theoretical and practical aspects of integration of strongly typed functional programming languages and relational databases. The main areas, covered by the work, include functional programming languages, relational databases and types.

Dissertation presents the results of corresponding research. The subject of the research was the analysis of the integration's aspects and the implementation possibilities. The dissertation topic is focused towards strongly typed functional programming languages, but in many segments of the research the general aspects of integration of programming languages and databases are discussed. One of the basic hypotheses was that the integration can be based on preserving the mutual orthogonality of integrated programming and query languages. Such approach should not bring into question basic properties of neither programming nor query language.

The paper presents important properties of functional programming languages and relational databases. It also presents the algorithms for the automatic type inference. The basic features of strongly typed programming language Wafl are presented. Wafl served as a platform for research and experimentation. Also, the type inference algorithm adapted to Wafl is presented. Different aspects of integration of programming languages and query languages are analyzed and discussed. Principles are defined, on which well-formed integration should be based. The identified basic principles are: abstraction, orthogonality and symmetry. Also, the concepts are recommended to achieve the integration according to the principles. The proposed concepts are the usage of query language in programming language, as well ad the usage of programming language in query language, the strongly typed integration, the unified transactional space, the orthogonality in respect to data types and the orthogonality in respect to aspects of the integration.

Each concept is developed and experimentally verified using the implementation of Wafl. The dissertation presents syntactic extensions for Wafl and SQL that allow the implementation of the presented concepts, as well as the semantics of the extensions. Static type checking algorithm for programs that use database is designed, based on the cooperation of programming language interpreter and database managing system.

The dissertation most important contributions include the recognition of the principles and concepts of well-formed integration of strongly typed programming and query languages, a completely new concept of host expressions, which are evaluated on the DBMS server side, an

implementation of support for complex programming language data types in the database with the appropriate type mapping, the concept and implementation of a unified transaction space and presented algorithms for static type checking of integrated solution. The results of this study may apply in both further research and practical use in the area of programming and query languages integration.

Keywords: functional programming, relational databases, types, strong type checking, implicit type inference, programming languages

Предговор

Савремени развој софтвера подразумева координацију и кооперацију већег броја различитих софтверских компоненти које чине један информациони систем. Посматрано из угла развоја софтвера, као основни градивни елементи неког рачунарског система се могу препознати програми, базе података и инфраструктура у оквиру које програми производе, употребљавају и одржавају податке. Сваки од ова три градивна елемента се по намени, начину развијања и употребе, теоријским основама и другим особинама значајно разликује од осталих.

Повезивање компоненти ради обезбеђивања одговарајуће функционалности се може остваривати на више различитих начина, али резултат практично увек мора да обухвати и програме и базе података и неке елементе инфраструктуре рачунарског система. Повезивање основних градивних елемената информационих система се обично одвија тако што се један од темеља прилагођава другом. Тако су, на пример, програмски језици често обликовани као императивни, да би се њихова семантика приближила императивној семантици хардверске инфраструктуре. Прилагођавање је ишло и у другом смеру, тако да је у неким случајевима хардверска основа прилагођавана конкретним програмским језицима. Слично томе, системи за управљање базама података су пролазили кроз различите развојне облике у покушајима да се на одговарајући начин прилагоде како програмским језицима тако и инфраструктури.

При таквим повезивањама најчешће се праве компромиси који угрожавају чак и неке основне особине компоненти које се повезују. Такви проблеми су у највећој мери последица неуспешног приближавања једног модела података и алгоритама другом. Програми представљају математичке структуре које почивају на различитим формализацијама појма алгоритма. У случају функционалних програмских језика те формализације имају статичку формалну природу. И релационе базе података имају формалну основу, али сасвим другачије природе. Оне почивају на релационом моделу података и концепту трансакционе обраде података. Сасвим другачију природу испољавају остали елементи инфраструктуре рачунарских система, који имају претежно процесну семантику.

Управо из уочених проблема и компромиса при постојећим начинима повезивања програмских језика и база података је настала и мотивација за истраживање из кога је потекао овај рад. У том контексту је најпре препознат појам *ортогоналног повезивања*, као повезивања основних градивних елемената информационог система на такав начин да се при томе не доводе у питање њихови независни појединачни квалитети. Препознато је и

питање: *Како* је потребно повезивати основне градивне елементе информационог система а да се при томе не доведу у питање њихови независни појединачни квалитети? У складу са интересовањима аутора, домен проблема је посматран пре свега у контексту повезивања строго типизираних функционалних програмских језика и база података. Ипак, где год се специфичан проблем не разликује много од општег случаја, истражене су и карактеристике општијег проблема повезивања.

У првом делу рада су представљене области обухваћене истраживањем. Поглавље 1 садржи преглед особина функционалних програмских језика. Истакнуте су разлике у односу на друге врсте језика, а посебно у односу на императивне програмске језике. Представљени су неки од најважнијих функционалних програмских језика. У поглављу 2 су наведене основне особине релационих система за управљање базама података. Представљени су основни елементи упитног језика *SQL*. Поглавље 3 је посвећено типовима и улози типова у програмским језицима и базама података. Поглавље 4 садржи приказ строго типизираног програмског језика *Wafl*, на коме су извођени сви експерименти и имплементације изведених конепата повезивања. Последње поглавље уводног дела садржи приказе неких од примера повезивања програмских језика и база података. Представљени су примери концептуалне и пуне интеграције.

Други део рада, *Строго типизирани функционални програмски језици и базе података*, садржи анализу проблема повезивања програмских језика и база података, са посебним акцентом на строго типизиране функционалне програмске језике. Средишњу уогу у дисертацији и овом делу има препознавање принципа и концепата *доброг* повезивања програмских језика и база података. Поглавље 6 се бави уопштеним проблемом повезивања програмских језика и база података. У поглављу 7 су размотрене разлике у системима типова у програмским језицима и базама података. Анализирани су могући начини превазилажења тих разлика. У поглављу 8 су препознати основни принципи на којима би требало да почива повезивање програмских језика и база података. Обликовани су основни концепти повезивања, са посебним акцентом на повезивање строго типизираних функционалних програмских језика. У поглављу 9 су изложени алгоритми за статичко извођење типова у сарадњи преводиоца за програмски језик и СУБП-а.

Трећи део рада, *Имплементација*, приказује како су представљени принципи и концепти имплементирани у случају програмског језика *Wafl*. Поглавље 10 је посвећено основним елементима повезивања програмског језика *Wafl* са релационим базама података. Представљени су упити, трансакције, матични изрази, библиотеке кода у бази података и друго. У поглављу 11 су изложени могући начини имплементирања представљених концепата. Поред представљања начина имплементирања који је спроведен у случају програмског језика *Wafl*, проблем се разматра и у ширим оквирима. Поглавље 12 садржи приказ алгоритама за проверавање типова програма који садрже елементе упитног и програмског језика, који су прилагођени специфичностима програмског језика *Wafl* и одговарајуће имплементације.

У четвртом делу рада се анализира сагласност представљене имплементације са концептима, као и сагласност концепата са усвојеним принципима. Анализирају се различити аспекти повезивања и њихове последице. Представљени су закључни ставови дисертације и могући смерови даљег истраживања.

Пети део чине додатни садржаји који доприносе бољем размевању рада. Поред индекса и списка литературе, ту су и пример примене алгоритма распознавања типова, имплементација корисничких функција за израчунавање матични израза на страни сервера и други додаци.

Дугујем велику захвалност свом ментору, др Ненаду Митићу, на пруженој подршци. Захваљујем се члановима комисије на пруженим саветима и наставницима Катедре за рачунарство и информатику на указаном поверењу.

Београд, мај 2009. године Аутор

Садржај

Спискови

Принципи повезивања

Концепти повезивања

Алгоритми

Слике

Програми

Табеле

Део I

Увод

1. Функционални програмски језици

1.1. Императивни и декларативни језици

Парадигме програмских језика представљају различите приступе решавању проблема уз помоћ рачунара и одговарајуће стилове и технике писања програма. У савременом рачунарству се препознаје велики број различитих парадигми. Међу најважније међусобно искључиве парадигме спадају императивна и декларативна парадигма.

Једна од основних карактеристика императивне парадигме је да програми представљају секвенце или сложеније структуре састављене од појединачних наредби, које на одређени начин мењају стање рачунарског система. Императивни алгоритми имају облик описа поступака израчунавања.

Императиван приступ је у самој основи фон-Најманове архитектуре рачунара, која је усмеравала развој дигиталних рачунара, практично од самог настајања дигиталних рачунара опште намене па све до данас. У основне концепте ове архитектуре рачунара спадају меморија и извршавање наредби. Меморија рачунара садржи податке којима програм оперише, и кодиране наредбе које управљају радом рачунара. Контролна јединица чита наредбе из меморије и управља њиховим извршавањем у сарадњи са аритметичко-логичком јединицом, меморијом и улазно излазним уређајима. У сваком тренутку рада рачунара, меморија (укључујући и све посебне врсте оперативне меморије који се појављују као саставни делови процесорских јединица и других делова система) садржи податке који чине стање система, а свака појединачна наредба коју процесорске јединице извршавају производи одговарајуће промене стања.

Императиван начин писања програма је тесно повезан са постојањем имплицитног стања система. Програми се састоје од елементарних операција које трансформишу то имплицитно стање. Појединачне операције производе елементарне промене стања, а програм као целина описује читав ток мењања неког почетног стања у завршно стање. Промена стања, која настаје извршавањем наредби програма, назива се бочним ефектом.

Због тога што се стање имплицитно преноси од једне до друге наредбе програма, за тачно дефинисање поступка трансформисања је неопходно експлицитно одређивање

редоследа извођења елементарних промена стања, тј. извршавања наредби програма. Последица је да се при писању императивних програма мора истовремено водити рачуна о два битна и релативно сложена аспекта: редоследу извршавања наредби и начину на који појединачне наредбе мењају имплицитно стање. Штавише, због имплицитног преношења стања, значење и улога делова програма се не могу разматрати само локално, већ неопходно у некој широј околини, која, у зависности од конкретних наредби и других околности, може обухватати свега неколико наредби али и цео програм.

Највећи број програмских језика који су данас у широкој употреби чине императивни програмски језици. Први виши програмски језици су настајали као надградње машинских и одговарајућих асемблерских језика процесора, задржавајући све основне особине императивног приступа обради информација. Развој императивних програмских језика текао је у смеру структурирања података и кода, најпре кроз обликовање тзв. структурног програмирања (програмски језици Паскал, Алгол, *C* и други), а затим и кроз интензиван развој објектно оријентисаног програмирања (програмски језици *Smalltalk*, *C++*, *Java* и други). Тесна повезаност основних особина императивних програмских језика и уобичајене императивне архитектуре савремених дигиталних рачунара је задржана кроз све генерације императивних програмских језика.

Потреба за усклађеношћу архитектуре хардвера и парадигме развоја софтвера постоји и јесте оправдана на најнижем нивоу, у случају развоја програма који непосредно управљају радом хардвера. Међутим, што се иде даље од хардвера и што су проблеми који се посматрају и решавају апстрактнији, то императивна парадигма развоја софтвера почиње да доноси мање погодности, а више проблема. Са порастом сложености проблема који се програмом решава, обично се повећава и сложеност стања које се преноси и мења императивним програмом. У таквим случајевима се значајно отежава писање, одржавање и проверавање исправности програма. Услед нарушене локалности стања и делова програма, формално проверавање исправности императивних програма је веома сложен проблем. Један од првих темељних критичких погледа на тесно повезивање проблема развоја софтвера и ниске архитектуре рачунарског система је дао Џон Бекус [Back1978].

Са развојем рачунарства се постепено развијала идеја да се програми могу писати на начин који допушта виши ниво апстрактности у односу на императивну парадигму. Другачији приступ писању програма омогућава *декларативна парадигма*. Док императивни програми описују *како* се нешто ради или рачуна, основна одлика декларативне парадигме је да програми описују *шта* је потребно израчунати или урадити.

Основне градивна јединица декларативних програмских језика представљају различити облици спецификација којима се формално описује жељени резултат. У декларативним језицима не постоји експлицитно описивање редоследа израчунавања ни имплицитно преношење и мењање стања. Због тога је за разумевање неког дела програма довољно познавање значења конкретне конструкције и разматрање њене улоге у непосредној околини, тј. у контексту у коме се експлицитно наводи у тексту програма. То ствара претпоставке за једноставније формално проверавање исправности декларативних програма него што је то случај са императивним програмима.

Најзначајније класе декларативних програмских језика представљају функционални и логички програмски језици. Најзначајнији представник логичких програмских језика је *Prolog*. Посебну категорију логичких језика чине различити системи за доказивање. У декларативне програмске језике се могу убројити и строго упитни подскупови неких језика база података, као што је, на пример, *SQL* [Date2003].

Поступак израчунавања програма написаних на декларативним програмским језицима је одређен имплицитно, семантиком програмског језика. На основу дате тачне спецификације проблема рачунарски систем „сам“ обавља израчунавање. Да би императивно заснован рачунарски систем могао да „сам“ израчуна резултат неопходно је да постоје софтверски алати који ће декларативне програме преводити на императивне или их непосредно израчунавати (интерпретирати).

Могуће је и направити рачунарски систем заснован на декларативном приступу. Развијено је више рачунарских система заснованих на декларативној парадигми. Они су махом примењивани у домену истраживања, уз релативно скромне практичне примене. Због тога такви системи нису остварили непосредан утицај на развој програмских језика. Међутим, остварили су значајан утицај на развој одређених технологија, као што су виртуалне машине, аутоматски сакупљачи отпадака и друге. Најзначајнији примери рачунара и процесора који су обраду информација изводили применом верзија функционалног програмског језика *Lisp* су фамилија *Maclisp*, која је развијана на *MIT*-у [Moon1979], и фамилија *Interlisp*, коју је развијала компанија *Xerox* [Teit1974].

1.2. Особине функционалних језика

Класу функционалних програмских језика чине декларативни програмски језици код којих се програми пишу записивањем израза које је потребно израчунати [Hudak1989]. Као и у случају других врста декларативних програмских језика, поступак израчунавања се не наводи експлицитно у програмима, већ се његово одређивање препушта преводиоцу или интерпретатору програмског језика.

Програми и потпрограми у функционалним програмским језицима се дефинишу веома слично уобичајеном начину дефинисања израза и функција у математици. Ова класа програмских језика је добила име по томе што је једини начин записивања мањих алгоритамских целина у програму управо дефинисање пресликавања једног скупа података у други, што потпуно одговара математичком појму функције. У употреби је и еквивалентан термин „апликативни програмски језици“. Атрибут „апликативни“ потиче од особине ових језика да је основна програмска конструкција *израз*, који у својој основи представља примену (енгл. *application*) неке функције на одговарајуће аргументе.

1.2.1. Основне особине

Скуп основних особина функционалних програмских језика може се одредити на више начина. Постоји више карактеристичних особина фунционалних језика [Hudak1989]. Све особине функционалних језика могу се систематизовати и поделити у *основне* и *изведене* особине.

Један минималан скуп основних особина функционалних програмских језика чине:

- имплицитно дефинисање тока израчунавања;

- одсуство имплицитног стања и
- записивање програма у облику израза који је потребно израчунати.

Декларативни програмски језици се одликују формалним начином апстраховања појмова алгоритма, програма и поступка решавања проблема уз помоћ рачунара. Семантика декларативних језика није везана за начин функционисања рачунарског система, па не постоји ни потреба за његовим подражавањем. Имплицитно дефинисање тока израчунавања и одсуство имплицитног стања су заједничке особине свих декларативних програмских језика.

Специфична особина функционалних програмских језика, по којој се разликују од осталих декларативних језика, је записивање програма у облику рачунских израза. Док се императивни програми „извршавају“, функционални програми се „израчунавају“. Основни циљ израчунавања сваког функционалног програма је добијање резултата, док циљ извршавања императивног програма може бити и израчунавање резултата, али је врло често циљ управо постизање неке промене стања рачунарског система. Ова разлика је веома важна и у значајној мери одређује примењивост (односно непримењивост) функционалних програмских језика у одређеним доменима.

Сложени изрази у функционалним програмским језицима се граде примењивањем функција на одговарајуће аргументе. Поред функција, у структури израза сличну улогу имају и различите операције и неке посебне синтаксне конструкције. Операције се обично представљају симболима оператора, који се употребљавају у инфиксној нотацији. У посебне синтаксне конструкције спадају, на пример, условни изрази. Међутим, ова разноликост на нивоу синтаксе се не преноси на ниво семантике, јер свим овим врстама сложених израза одговара једна иста семантика функције. Због тога се у даљем тексту термин „функција“ употребљава у ширем смислу, тако да обухвата и све посебне случајеве функција. Од тога се одступа само у случајевима када је реч о синтакси и конкретним синтаксним конструкцијама.

1.2.2. Изведене особине

Неке од непосредних последица наведених основних особина функционалних програмских језика се често убрајају у основне особине, јер су заједничке за све функционалне програмске језике. Ипак, овде се ради прецизности оне препознају као *изведене* заједничке особине функционалних језика.

Најважније изведене особине функционалних програмских језика су:

- непостојање бочних ефеката;
- дефинисање потпрограма у облику функција;
- важење референцијалне транспарентности;
- употреба рекурзије као јединог средства за описивање „понављања“ и
- имплицитно управљање ресурсима.

Термин „бочни ефекат“ означава промене стања које настају као последица извршавања (или израчунавања) неке конструкције програмског језика или дела програма. Бочни ефекти су нераскидиво везани за императивни концепт писања програма који мењају стање система. Како у основне приципе функционалног

програмирања спада и одсуство имплицитног стања, непосредна последица је да нема ни бочних ефеката.

Уобичајени облици потпрограма у императивним програмским језицима су процедуре и функције. Основни посао процедура је да мењају стање система на одговарајући начин, док функције, поред тога што могу да мењају стање система, израчунавају и неки резултат. У функционалним програмским језицима постоје само функције, као једини облик потпрограма. Штавише, оне имају само један и сасвим прецизно одређен задатак – да израчунају неки резултат на основу датих аргумената. Функције у функционалним програмским језицима не могу да мењају стање система. Тачна семантика функције је одређена искључиво изразом којим је функција дефинисана. Захваљујући томе, логичка верификација функције је могућа без анализирања ширег контекста, само на основу њене дефиниције.

Одсуство стања програма и рачунарског система има за непосредну последицу да поновљено израчунавање истог израза увек даје исти резултат. Штавише, ако се исти подизраз (без слободних променљивих) појављује и израчунава на више места у програму, он мора имати исти резултат. Ова особина функционалних програмских језика се назива „референцијална транспарентност“. Референцијална транспарентност пружа пуну сигурност при поновљеној употреби већ написаних израза, функција и програма, али и представља основу за увођење оптимизација у процес израчунавања.

У пракси, референцијална транспарентност је обично само делимично примењена. Практично сваки програмски језик обухвата неке конструкције програмског језика које јесу референцијално транспарентне, али и неке које нису у потпуности сагласне са овим принципом. На пример, функције које израчунавају псеудо-случајан број или тренутно време или датум нису транспарентне. Сложенији примери су изрази који читају улазне податке са тастатуре или приступају садржају датотеке или базе података. Чак и ако програмски језик није потпуно референцијално транспарентан, он може омогућавати писање транспарентних израза и програма.

Одсуство експлицитног описивања тока израчунавања има за последицу и одсуство наредби за понављање извршавања одређеног сегмента програма. Одговарајући концепт у функционалним програмским језицима је рекурзија. Рекурзија представља формално математичко средство за записивање функционалних зависности на које су примењиви принципи математичке индукције. Основна разлика између рекурзије и понављања је у стриктно формалној природи рекурзије, са једне, и јасно изражене процесне природе понављања са друге стране. У том смислу, ова два појма су тешко упоредива. Ипак, примена рекурзије у функционалним програмима представља практични еквивалент примени понављања у императивним програмима. Рекурзија се примењује како за описивање сложенијих облика израчунавања, тако и за описивање сложених структура података. Логичка верификација рекурзије се непосредно своди на математичку индукцију, због чега је далеко једноставнија од потенцијално веома сложене верификације наредби понављања.

Још једна веома важна последица одсуства имплицитног стања је да у функционалним програмским језицима није могуће остварити експлицитно управљање ресурсима. Наиме, да би се ресурсима могло експлицитно управљати, неопходно је памтити и мењати њихово стање, а како то у функционалним програмима не може да се оствари, није могуће ни експлицитно управљање ресурсима.

Имплицитно управљање ресурсима је најуочљивије у контексту управљања радном меморијом. Израчунавања са сложеним структурама података и у императивним и у функционалним језицима имају потенцијално веома велике захтеве по питању заузимања и ослобађања радне меморије. Међутим, док се код императивних језика то обавља у великој мери (или чак потпуно) експлицитно, функционални програми једноставно морају да користе искључиво имплицитно управљање меморијом.

При аутоматском управљању меморијом, имплементација програмског језика имплицитно заузима нове меморијске локације, у складу са семантиком израчунавања израза у конкретном језику, док се за ослобађање меморије која више није потребна употребљавају аутоматски сакупљачи отпадака. Аутоматски сакупљачи отпадака су дуго били специфичност функционалних програмских језика. Напредак алата за развој софтвера је довео до тога да је и значајан број императивних програмских језика почео да користи аутоматске сакупљаче отпадака (*Java*, *C#* и други).

1.2.3. Операције са функцијама

У функционалним програмским језицима функције су обично *грађани првог реда*. Функције се могу користити као аргументи функција или израчунавати као резултати израза и функција. Функције које за аргументе и/или резултате имају функције називају се *функције вишег реда*. Писање израза који оперишу функцијама и дефинисање функција вишег реда најчешће нису синтаксно сложенији него употреба других основних типова података. Наравно, као последица потенцијално високог нивоа апстракције, семантика таквих израза може бити веома сложена.

Делимично израчунавање

Једна од основних операција са функцијама је *делимично израчунавање*. Делимично израчунавање је операција којом се функција примењује на мањи број аргумената него што је одређено њеном дефиницијом. Делимичним израчунавањем се добија нова функција, која има мање аргумената него претходна, а чијим се даљим примењивањем практично допуњавају недостајући аргументи полазне функције. Делимично израчунавање се у општем случају односи на примену функције на било које, али не све потребне аргументе. У употреби је и термин *незасићена* (или *непотпуна*) примена. Техника делимичног израчунавња је веома значајна у области развоја софтвера [Cons1998]. Делимично израчунавање је један од најчешће коришћених начина за дефинисање израза који израчунавају функције.

Формално, ако је f функција са n аргумената, делимично израчунавање у облику примене на i-ти аргумент се може дефинисати као:

$$\pi_i(f, a_i) \stackrel{def}{=} f^{[x_i=a_i]}$$

$$f^{[x_i=a_i]}(x_1,...,x_{i-1},x_{i+1},...,x_n) \stackrel{def}{=} f(x_1,...,x_{i-1},a_i,x_{i+1},...x_n)$$

На основу ове дефиниције може се дефинисати оператор π, за делимично израчунавање применом функције f на један аргумент:

$$\pi(f,i,x) \stackrel{def}{=} \pi_i(f,x)$$

Вишеструком применом оператора π се добија уопштено делимично израчунавање.

Посебан случај делимичног израчунавања представља примена тзв. *Каријевих функција* (енгл. *Curryed functions*). Основна идеја је да се свака функција, која има више аргумената, изрази у облику функције од једног аргумента чији је резултат функција од преосталих аргумената. Резултат таквог приступа је да свака примена функције на више аргумената представља композицију делимичних примена на први аргумент.

Већи број функционалних програмских језика подржава Каријеве функције, док само мањи број језика подржава опште делимично израчунавање. Програмски језици који подржавају Каријеве функције, али не и уопштено делимично израчунавање, обично имају синтаксу примене функције без навођења заграда око аргумената, јер се свака функција (бар на семантичком нивоу) примењује увек на само један аргумент, а ако резултат такве примене представља функцију, онда се она потенцијално примењује на следећи аргумент, и тако даље. Формално, примена функције f на један аргумент се дефинише као:

$$\texttt{f x} \overset{def}{=} \texttt{f}^{[x_1=x]}$$

На пример, ако је функција `sub` дефинисана са:

```
sub x y = x-y
```

тада се, у случају примене Каријевих функција, у изразу `sub 5 3` функција `sub` најпре примењује на први аргумент – број 5. Тиме се израчунава унарна функција, која аргумент слика у разлику броја 5 и аргумента. Затим се тако добијена функција примењује на аргумент 3, чиме се добија вредност 2. Значи, ток израчунавања је:

```
sub 5 3
 --> (sub 5) 3
 --> 2
```

Изрази `(sub 5) 3` и `sub 5 3` су међусобно еквивалентни. Тип функције `sub` би могао да се представи као: `Int -> Int -> Int`, при чему се претпоставља да је оператор пресликавања типова десно асоцијативан, па је претходни тип еквивалентан типу `Int -> (Int -> Int)`.

У програмским језицима који подржавају опште делимично израчунавање, оно се обично остварује применом оператора `bind` за везивање аргумената. Семантика оператора `bind` за везивање тачно једног аргумента је еквивалентна представљеном формално дефинисаном оператору π. На пример, ако је оператор `bind` дефинисан тако да се са `bind(f,i,a)` као `i`-ти аргумент функције `f` везује вредност `a`, онда су сви наредни изрази међусобно еквивалентни:

```
sub(5,3)
bind(sub,1,5)(3)
bind(sub,2,3)(5)
bind(bind(sub,2,3),1,5)
bind(bind(sub,1,5),1,3)
2
```

Ламбда изрази

Због важне улоге и честе употребе релативно једноставних функција у различитим контекстима ради изградње сложенијих функција, већина функционалних програмских језика омогућава да се дефинишу неименоване функције. Тиме се растерећује простор имена, који би иначе био затрпан већим бројем имена функција које се једнократно употребљавају.

Неименоване функције се обично називају *ламбда изрази* (или *ламбда функције*) јер се дефинишу по узору на апстракције ламбда рачуна. Ламбда рачун је једна од најважнијих формализација појма алгоритма [Bare2000]. Основни елементи ламбда рачуна су *апстракција* и *апликација*. Апстракција је израз облика *λx.exp*, који представља функцију која пресликава *x* у *exp*. Апликација је примена апстракције на дати аргумент. Облик у коме се у програмским језицима записују неименоване функције обично има сличности са записом апстракција у ламбда рачуну.

У наредном примеру је илустрована употреба ламбда функције. Функције `kvadrat1` (дефинисана помоћу именоване функције `kub`) и `kvadrat2` (дефинисана помоћу ламбда функције еквивалентне са функциојом `kub`) су међусобно еквивалентне:

```
kub x = x*x*x;
kvadrat1( lst ) = map( lst, kub );
kvadrat2( lst ) = map( lst, \x -> x*x*x );
```

Операције са функцијама у другим програмским језицима

Операције над функцијама нису више специфичност функционалних програмских језика. Већи број императивних језика омогућава да се као аргументи и/или резултати израза и потпрограма употребљавају потпрограми. Међутим, због природе функционалних језика се ове операције далеко боље слажу са осталим елементима језика, па су због тога и једноставније за употребу.

Значајан број савремених императивних језика располаже напредним могућностима рада са функцијама, као што су делимично израчунавање, Каријеве функције и ламбда функције. Скоро по правилу, то се омогућава развојем библиотека заснованих на релативно сложеним апстракцијама, које почивају на концепту генеричког полиморфизма, као у случају бибилиотека *Loki* [Alex2001] и *Boost* [Abra2004] за програмски језик *C++*, или библиотеке *Mochikit* [Mooc:2008] за *JavaScript* [Hein1997]. Актуелне верзије програмских језика *Java* и *C#* се одликују подршком за шаблоне по узору *C++*, па се сада и у њима могу имплементирати сличне библиотеке.

1.2.4. Стриктна и нестриктна семантика

Поступак израчунавања фунционалних програма, па тиме и појединачних израза, се у функционалним програмским језицима не описује експлицитно, али се може претпоставити на основу семантике програмског језика. Међутим, док је концепт израза у основи сасвим формалан и заједнички за све функционалне програмске језике, дотле се имплицитно одређивање редоследа може разликовати.

Постоје два основна могућа редоследа израчунавања израза: апликативни и нормални. Апликативни редослед израчунавања израза подразумева да се сви аргументи функције израчунавају пре него што започне израчунавање саме функције.

Овај начин израчунавања се назива и „унутрашњи“ (енгл. *innermost*), или „од листова према корену“, јер се најпре израчунавају најдубљи унутрашњи делови израза (који при представљању израза у облику дрвета имају улогу листова).

У случају нормалног редоследа израчунавања, израчунавању тела функције се приступа без претходног израчунавања аргумената. Аргументи се израчунавају тек када то буде неопходно при израчунавању тела функције. Овај начин израчунавања се назива и „спољашњи“ (енгл. *outermost*) или „од корена према листовима“, јер се најпре израчунавају спољашњи делови израза (који при представљању израза у облику дрвета имају улогу корена), а унутрашњи се израчунавају онда када то постане неопходно за даље израчунавање.

На пример, ако су функције `f` и `g` дефинисане као:

```
f(x) = x*x + x + 2;
g(x,y) = x+y;
```

онда се применом апликативног редоследа израчунавања долази до резултата израза `f(g(2,3))` низом корака:

```
f(g(2,3))
 --> f(2+3)
 --> f(5)
 --> 5*5+5+2
 --> 32
```

док се у случају нормалног редоследа израчунавања до резултата долази корацима:

```
f(g(2,3))
 --> g(2,3)*g(2,3)+g(2,3)+2
 --> (2+3)*(2+3)+(2+3)+2
 --> 5*5+5+2
 --> 32
```

Према начину израчунавања израза функционални програмски језици се деле на *стриктне* и *нестриктне* језике. Стриктни језици су они који семантику израза дефинишу на основу апликативног редоследа израчунавања, а нестриктни су они који је дефинишу на основу нормалног редоследа израчунавања. У пракси је могуће комбиновање ова два начина израчунавања, о чему ће бити речи нешто даље у овом одељку.

Лењо и вредно израчунавање

Нестриктна семантика обезбеђује две значајне карактеристике израчунавања израза:

- израчунавање сваког конкретног подизраза се одлаже све до тренутка када његова вредност буде неопходна за израчунавање резултата програма и
- не израчунавају се читави изрази већ само они подизрази од којих зависи израчунавање резултата.

Иако нестриктна семантика омогућава да се заобилази израчунавање подизраза који не утичу на резултат, примена нормалне редукције је ипак често мање ефикасна од апликативне редукције. Један проблем је у томе што алгоритам нормалне редукције захтева више операција са изразима док алгоритам апликативне редукције више ради са израчунатим вредностима, што је потенцијално ефикасније.

Други проблем је у томе што нормална редукција има за последицу да се један исти подизраз може израчунавати више пута, што се не дешава у случају апликативне редукције. На пример, у претходном примеру са израчунавањем израза `f(g(2,3))`, подизраз `g(2,3)` се израчунава само једанпут у случају апликативног, а чак три пута у случају нормалног редоследа израчунавања.

На основу референцијалне транспарентности следи да сва поновљена израчунавања једног истог подизраза морају увек имати исти резултат. Због тога је могуће у имплементацију поступка рачунања увести наредну оптимизацију, која представља додатну карактеристику израчунавања:

- сваки подизраз се рачуна највише једанпут, а сваки следећи пут се уместо израчунавања само употребљава већ израчуната вредност.

Нестриктно израчунавање са наведном додатном карактеристиком се назива *лењо израчунавање*. Као пуна супротност лењом израчунавању, израчунавање у апликативном редоследу, тј. примена стриктне семантике, обично се назива *вредним израчунавањем*.

Чак и са наведеном оптимизацијом, поређење ефикасности лењог и вредног израчунавања није могуће уопштено анализирати, јер значајно зависи од контекста. Са једне стране, лењо израчунавање омогућава да се до резултата дође уз потенцијално мање израчунавања, али, са друге стране, оно уводи додатну сложеност у имплементацију, чиме се израчунавање у одређеној мери успорава. У пракси, сваки од начина израчунавања је у неким случајевима ефикаснији, а у неким другим неефикаснији.

Бесконачне структуре података

Један исти референцијално транспарентан израз (или подизраз) у било ком програму и на било ком месту у програму увек мора да има исту вредност. Одатле непосредно следи да, ако важи:

1. да су израз функционалног програмског језика и сви његови подизрази добро дефинисани и израчунљиви и
2. при избору редоследа израчунавања се поштује асоцијативност операција,

тада изабрани редослед израчунавања нема утицаја на вредност резултата израчунавања израза. Самим тим, ако су испуњени наведени услови, тада ће апликативан и нормалан редослед израчунавања доводити до истог резултата.

Разлика, међутим, настаје у случају израза који нису израчунљиви, јер неке класе израза са неизрачунљивим подизразима могу да се израчунају применом нормалног али не и апликативног редоследа редукције израза. На пример, нека је функција `f` неизрачунљива у `x` (на пример, дефинисана је са бесконачном рекурзијом) а функција `g` јесте израчунљива у `x`. Тада следећи израз не може да се израчуна применом апликативног редоследа израчунавања, али може да се израчуна применом нормалног редоследа израчунавања:

```
(\a,q,w : if a<0 then q else w)( 3, f(x), g(x) )
```

Апликативни редослед израчунавања би започео израчунавањем подизраза `f(x)` и `g(x)`. Како функција `f` није дефинисана у `x`, израчунавање би произвело грешку. Са друге стране, у случају нормалног редоследа израчунавања, најпре би се проверио услов,

па би се онда израчунавала потребна грана и, коначно, само они аргументи који се у њој употребљавају – у овом случају израчунљив израз `g(x)`:

```
(\a,q,w : if a<0 then q else w)( 3, f(x), g(x) )
 --> if 3<0 then f(x) else g(x)
 --> g(x)
 --> ...
```

При нормалном редоследу израчунавања је практично омогућено да се у програмима употребљавају чак и неизрачунљиви изрази, све док се не инсистира на њиховом пуном израчунавању. Пажљивом употребом неизрачунљивих израза добија се потпуно нов квалитет при писању програма – могућност дефинисања и коришћења *бесконачних структура података*[1]. Бесконачне структуре података почивају на примени отворене (тј. бесконачне) рекурзије за дефинисање листа са пребројиво много елемената.

Стварна величина декларативно бесконачних структура података је ограничена имплицитно, ресурсима рачунарског система. Посебан квалитет је да се о томе не мора водити рачуна при њиховом дефинисању, већ само при употреби, јер се бесконачне структуре при израчунавању програма имплицитно израчунавају само у оној мери у којој се употребљавају.

На пример, у нестриктном програмском језику би се листа свих природних бројева могла дефинисати на следећи начин:

```
prirodni = prirodniOd(1);
prirodniOd(n) = n : prirodniOd(n+1);
```

Сваки покушај израчунавања листе природних бројева у стриктном програмском језику би се завршио неуспехом, јер би стриктна семантика налагала да се израчунају сви елементи листе. Са друге стране, нестриктна семантика омогућава да се листа остави у неизрачунатом или делимично израчунатом облику, па је оваква дефиниција у потпуности исправна. Све док се при употреби бесконачне структуре не инсистира на долажењу до њеног „краја“, практично се употребљава само коначан број почетних елемената листе и нормалан редослед израчунавања ће дати резултат[2].

На пример, нека је изразом `subList(prirodni,0,30)` одређена листа првих 30 природних бројева. Тај израз није израчунљив применом апликативног, али јесте израчунљив применом нормалног редоследа израчунавања. Ако је употреба листе `prirodni` у датом изразу уједно и једина у програму, онда ће листа природних бројева бити израчуната највише до 30. елемента, док ће наставак листе остати у неизрачунатом облику, као апстракција `prirodniOd(31)`.

Са друге стране, израз `length(prirodni)` рачуна дужину листе природних бројева, па је очигледно да ће бити неизрачунљив, без обзира на одабран редослед израчунавања. Да би се избројали сви елементи листе, чак и у случају лењог израчунавања је неопходно доћи до њеног краја, што у случају бесконачне листе није остварив о.

---

[1] Овај термин је ушао у употребу иако није сасвим исправан. Структуре података са којима се ради у коначној меморији и коначном времену заправо могу бити само *декларативно* бесконачне.

[2] Наравно, уз претпоставку да је на располагању довољно радне меморије и процесорског времена.

Однос стриктне и нестриктне семантике

Стриктна семантика је интуитивно ближа начину на који човек приступа израчунавању израза. Сваки аргумент фунције се израчунава тачно једанпут, а затим се израчунате вредности аргумената употребљавају при израчунавању резултата функције. Стриктна семантика је и једноставнија за имплементацију, јер се при израчунавању рукује само вредностима и непроменљивом структуром израза.

Са друге стране, нестриктна семантика је погоднија за формалне примене, јер је еквивалентна ламбда изразима, који представљају једну од формализација појма алгоритма. Она је сложенија за имплементацију, јер се осим вредностима и почетним изразом, при израчунавању динамички ради са изразима који се праве током процеса израчунавања. Док се применом стриктне семантике сваки аргумент функције израчунава тачно једанпут, у случају нестриктне семантике се може догодити да се неки аргументи израчунавају више пута, али и да се уопште не израчунавају.

Нестриктни изрази у стриктним језицима

Ако разматрање семантике спустимо на ниво конкретних функција и операција, можемо приметити да већина стриктних програмских језика, како функционалних тако и императивних, располажу неким елементима који су нестриктни.

Готово сви савремени стриктни програмски језици дефинишу примитивне логичке операторе на нестриктан начин. У тим језицима, ако се израчунавањем првог аргумента логичке конјункције добије логичка вредност „нетачно“, тада се други аргумент не израчунава, јер је већ познато да је резултат „нетачно“. Слично, ако се израчунавањем првог аргумента логичке дисјункције добије логичка вредност „тачно“, тада се други аргумент не израчунава, јер је већ познато да је резултат „тачно“.

Одлука о начину израчунавања логичких оператора `or` и `and` је углавном необавезне природе. Аутори језика који бирају нестриктну семантику то чине пре свега ради повећавања ефикасности израчунавања. Међутим, избор нестриктне семантике има и много шири значај. Прва значајна последица је проширивање области дефинисаности израза, који имају потенцијално недефинисане подизразе. На пример, наредни израз је дефинисан и не производи грешке при израчунавању у условима нестриктне семантике оператора `and`, али није дефинисан и производи грешку при израчунавању у случају стриктне семантике оператора `and` и вредности променљиве `x=0`:

```
x>0 and 5/x>20
```

Нестриктне дефиниције логичких оператора се могу формално свести на примену нестриктног условног израза `if`:

```
x and y = if x then y else false
x or y = if x then true else y
```

Нестриктни условни израз `if` подразумева да се, у зависности од вредности услова, израчунава тачно једна од алтернативних грана. Еквивалент условном изразу `if` постоји у већем броју императивних програмских језика. Међутим, док би нестриктност тог израза у императивним језицима можда и могла бити предмет избора, у случају функционалних језика такав избор не постоји – у свим функционалним програмским језицима, како нестриктним тако и стриктним, условни израз мора да има нестриктну семантику.

Другачији приступ, са евентуалним израчунавањем услова и обе гране пре одлучивања о резултату, неизоставно би имао за последицу неупотребљивост како конкретног условног израза, тако и самог језика. Проблем је у томе што се рекурзија, као једна од основних конструкција функционалних језика, заснива на нестриктној примени условног израза. У случају стриктне семантике условног израза чак и најједноставна рекурзија би постала отворена, а тиме и неизрачунљива.

1.2.5. Чисто фунционални програмски језици

Већ је истакнуто да је једини циљ израчунавања функционалних програма управо израчунавање резултата. За разлику од њих, императивни програми могу за циљ извршавања да имају и промену стања рачунарског система. Ова разлика је веома важна и у значајној мери одређује области примењивости функционалних програмских језика.

Опсег проблема који се решавају помоћу рачунара је веома широк. Штавише, свакодневно се проширује. Ипак, сви проблеми који се решавају помоћу рачунара могу се поделити на две основне категорије:

- рачунски проблеми и
- оперативни проблеми.

Рачунски проблеми су проблеми при чијем је решавању једини циљ извођење неког израчунавања, тј. израчунавање одговарајућег резултата. Они се могу успешно решавати применом функционалних програмских језика[3]. Штавише, особине функционалних програмских језика им дају неке значајне предности у решавању рачунских проблема у односу на императивне програмске језике.

Оперативни проблеми су они при чијем је решавању основни циљ да се остваре одговарајуће промене стања рачунарског система. Променом стања рачунарског система се може утицати на рад различитих уређаја који су повезани са рачунарским системом. Промена стања представља управо и једини начин на који рачунари могу нешто да „ураде“ у спољном свету. Међутим, основне особине функционалних програмских језика имају за непосредну последицу да функционални програмски језици не могу непосредно да послуже за решавање оперативних проблема. У програмским језицима који доследно поштују основне принципе функционалног програмирања не постоје конструкције које омогућавају интеракцију са стањем програма или рачунарског система. Последица је да природа оперативних проблема практично захтева примену императивних конструкција програмских језика.

Тежња за проширивањем домена примене функционалних програмских језика има за последицу да се при обликовању већег броја функционалних програмских језика праве одређени компромиси у односу на основне принципе функционалног програмирања, а са циљем омогућавања *умерене* подршке императивном начину програмирања. Такви компромиси су усмерени на увођење одређеног вида имплицитног стања и бочних ефеката, којима се то стање може мењати. На тај начин се непосредно угрожава референцијална транспарентност, а тиме и већи број других особина функционалних језика које представљају последице референцијалне транспарентности.

[3] Осим, наравно, ако је проблем сам по себи нерешив.

Један начин да се одржи функционална природа програма а проблем делимично превазиђе јесте уградња програма у извршно окружење које самостално изводи промене стања на основу резултата израчунавања програма. У таквим окружењима се дефинише одговарајућа семантика „покретања“ програма тако да програм израчунава „ново стање“ као резултат, а затим се, на крају израчунавања програма, независно од самог поступка израчунавања, старо стање система замењује новим. То у пракси значи да се све промене које је потребно извршити акумулирају у резултату израчунавања и на крају се заједно примењују. Проблем са оваквим приступом је да није сасвим погодан ни за масовне обраде ни за интерактиван рад. Масовне обраде подразумевају промене великог броја података, што је врло неефикасно акумулирати. Са друге стране, интерактиван рад се ограничава на интеракције између два покретања програма (било једног истог или различитих програма), што је далеко од пуне интерактивности.

Програмски језици који су доследно декларативни и у потпуности поштују принципе функционалног програмирања називају се *чисто функционалним програмским језицима*. Све до овог места термин *функционалан програмски језик* је у овом раду употребљаван у ужем смислу, тј. у значењу термина *чисто функционалан програмски језик*. У даљем тексту ће, међутим, под *функционалним програмским језицима* бити подразумевани и они језици који принципе функционалног програмирања не поштују у потпуности.

У функционалним језицима који нису чисто функционални, обично се могу препознати неке конструкције језика и одговарајуће класе израза које јесу чисто функционалне. Део језика који чине само такве конструкције назива се и чисто функционалним подскупом језика. Може се приметити да и неки императивни програмски језици садрже чисто функционалне подскупове језичких конструкција, али такви подскупови обично не обухватају неке од најважнијих елемената језика[4].

Редослед израчунавања израза

Као што је раније показано, ако важи принцип референцијалне транспарентности, тада редослед израчунавања израчунљивог израза не утиче на резултат. Међутим, ако се ради о језику у коме референцијална транспарентност не важи у потпуности, тада је неопходно да семантика језика одређује прецизније редослед израчунавања израза.

Без обзира на начин формалног дефисања редоследа израчунавања подизраза у неком конкретном језику, већ сама чињеница да се он формално дефинише непосредно доводи у питање саму природу нестрикних језика. Ако би, на пример, било дефинисано да се аргументи функција увек рачунају редом, од првог до последњег, онда би, у тренутку када неки аргумент постане неопходан за даље израчунавање, увек прво било неопходно израчунати све претходне аргументе. Тиме семантика језика у значајној мери поприма стриктне особине, штавише уз нимало очигледно закључивање о томе какав ће редослед израчунавања израза бити употребљен у неком конкретном случају. Због тога су нестриктни језици обично доследнији у поштовању референцијалне транспарентности, а тиме и осталих принципа функционалног програмирања.

Са друге стране, у стриктним језицима прављење сличних компромиса не производи исте проблеме, јер је редослед израчунавања и иначе предефинисан. Основна разлика је

[4] На пример, чак и програмски језик *C* има одговарајући чисто функционалан подскуп. Међутим, он је сувише узак да би се у пракси могао значајније употребљавати.

што се правило „прво аргументи па функција“ допуњава још и одговарајућим правилом о редоследу израчунавања аргумената.

Чисто функционалан приступ императивним проблемима

Иако „чисто функционално програмирање“ и „промена стања“ представљају два међусобно искључива концепта, постоје неки чисто функционални приступи императивним проблемима. Заправо, одговарајућа решења почивају или на прављењу одређених компромиса, или на чисто функционалном апстраховању концепата стања и промене стања.

Апстраховање стања и промене стања почива на замени имплицитног стања експлицитним стањем, у облику у коме је прихватљиво за чисто функционалне програмске језике. Све конструкције језика које би требало да читају или мењају стање, имају за експлицитан аргумент почетно стање система, а као део резултата израчунавају евентуално измењено стање система. Експлицитним преношењем стања се обезбеђује важење свих принципа функционалног програмирања, али се програмер оптерећује сложенијом синтаксом и семантиком одговарајућих делова кода. Ипак, даљим унапређивањем синтаксе и апстраховањем одговарајућих типова се може обезбедити да експлицитно стање ипак не мора да се експлицитно записује у читавом тексту програма, већ само на почетку одговарајућих сегмената кода.

Једна од најзначајних апстракција императивног програмирања на чистом функционалном програмском језику је приступ заснован на *монадама*, имплементиран у програмском језику *Haskell* [Jones:1993]. Монаде омогућавају да *Haskell* остане чисто функционалан а да истовремено подржава интерактиван рад и рад са трајним подацима. Монаде обезбеђују одговарајућу семантику за експлицитно преношење стања, док се одговарајућа синтакса може обликовати тако да експлицитно преношење стања не оптерећује код, тј. да синтакса самог преношења буде делимично имплицитна. Резултат је да се, зависно од проблема, на *Haskell*-у могу писати програми коришћењем како функционалног тако и императивног приступа програмирању, а да написани код програма у оба случаја у својој основи јесте чисто функционалан.

Због тога што монаде представљају довољно апстрактан концепт за моделирање и функционалног и императивног израчунавања, оне се често употребљавају и као основа за дефинисање семантика програмских језика [Tolm2003, Fili2007]. Међутим, омогућавањем "чисто-функционалног" императивног програмирања проблеми нису решени, јер су анализирање и верифковање таквог кода скоро једнако сложени као и у случају обичног императивног кода.

1.2.6. Формална семантика

Без обзира на изабран начин израчунавања израза, због одсуства стања и управљања редоследом извршавања наредби програма, семантика чисто фунционалних језика се може формално дефинисати на једноставнији начин него што је то случај са императивним програмским језицима.

Формално дефинисање синтаксе и семантике језика представља један од основних предуслова за аутоматизацију поступака валидације и верификације програма. У случају чисто функционалних програмских језика је за разумевање неког дела програма довољно познавање значења конкретне конструкције и разматрање њене улоге у

непосредној околини, тј. у контексту у коме се експлицитно наводи у телу програма. Не постоји потреба да се разматра неки шири контекст, јер референцијална транспарентност обезбеђује да посматрани израз (или функција) даје исти резултат у свим условима. Није потребно ни разматрање стања програма, јер оно не постоји. Тиме се омогућава локално разумевање елементарних делова програма и њихове улоге у самом програму, што значајно поједностављује логичко закључивање о њиховој исправности, а тиме и писање исправних програма и њихово аутоматско верификовање.

Доказивање коректности програма се обично састоји из два корака: из доказивања делимичне коректности (тј. доказивања да је резултат исправан у свим случајевима када израчунавање програма стаје и даје резултат) и доказивања да програм стаје за све допуштене вредности аргумената. Због доследне декларативне природе чисто функционалних програмских језика, први корак је обично релативно једноставан: како се програмом описује *шта* је потребно израчунати, а не *како* се израчунавање изводи, доказивање делимичне коректности се своди на проверу да ли, под условом да аргументи функције задовољавају неке дате предуслове, аргументи и резултат функције задовољавају дате постуслове.

На пример, ако би у оквиру дефиниције фукције `korenJednacine` (која рачуна један корен квадратне једначине) предуслов и постуслов били дефинисани на следећи начин:

```
korenJednacine(a,b,c) =
    [precondition: b*b >= 4*a*c ]
    [postcondition: a*result*result + b*result + c = 0]
    (-b + sqrt(b*b-4*a*c))/2/a;
```

тада би се провера коректности свела на проверавање да ли, при испуњеном предуслову, важи постуслов:

```
a * ((-b + sqrt(b*b-4*a*c))/2/a) * ((-b + sqrt(b*b-4*a*c))/2/a)
+ b * ((-b + sqrt(b*b-4*a*c))/2/a)
+ c = 0
```

Иако проверавање таквих услова у општем случају није тривијално, ред сложености проблема који се овде појављују је значајно нижи него при проверавању коректности императивних програма, јер је у њиховом случају, поред провере задовољености услова, потребно још и пратити ток извршавања наредби и промена стања.

Други део провере коректности представља провера да ли програм стаје за све допуштене вредности аргумената. У случају функционалних програмских језика ова провера се своди на проверавање да ли су исправни домени функција и да ли су све употребљене рекурзије исправно дефинисане (тј. да нису отворене) [Giesl1995]. Штавише проблем може да се сведе не само на ниво појединачних функција већ и на ниво делова израза [Coll1994]. Такође, и ова врста провера је по својој природи једноставнија него одговарајући проблеми у случају императивних језика.

Све наведене погодности и олакшице при доказивању коректности стоје не само у случају чисто функционалних језика, већ и у случају чисто функционалних израза у осталим функционалним језицима. У случају делова програма у којима се користе елементи програмског језика који нису чисто функционални, проблем се усложњава и није значајно једноставнији него у случају императивних програмских језика.

1.3. Представници функционалних програмских језика

У наредним одељцима укратко су представљене основне карактеристике тријy најважнијих представника функционалних програмских језика. *Lisp* је представљен као најстарији и дуго најпознатији представник функционалне парадигме, а *ML* и *Haskell* као два савременија функционална програмска језика.

1.3.1. Lisp

Програмски језик *Lisp* [McCa1978] је први функционалан програмски језик. Осмислио га је *John McCarthy* 1958. године. Прву спецификацију језика је објавио 1960. године [McCa1960]. Прву имплементацију интерпретатора је развио *Steve Russell*. Први потпун компилатор, развијен на самом *Lisp*-у, написали су *Tim Hart* и *Mike Levin* на *MIT*-у 1962. године.

Колико су активности на програмском језику *Lisp* биле пионирске, показује и чињеница да је у то време постојао само још један програмски језик који је и данас у употреби. Реч је о програмском језику *Fortran*[5] [ANSI:1978]. Остали језици настали у то време углавном су имали кратку историју. Један од језика чији су концепти делимично утицали и на настанак *Lisp*-а је *IPL* (енгл. *Information Processing Language*), који се сматра првим језиком у домену вештачке интелигенције.

Lisp је увео неколико значајних новина у област програмских језика:

- употреба условног израза и његова употреба при дефинисању рекурзивних функција;
- употреба листе и операција вишег реда над листама као основа сваког рада са сложеним структурама података;
- имплицитно управљање меморијом путем употребе аутоматског сакупљача отпадака и функција за конструисање листе и приступање њеним деловима;
- *S*-изрази као апстрактна синтакса за истоветно представљање програма и података – *Lisp* је први хомоиконичан програмски језик.

Једна од најважнијих особина *Lisp*-а је његова изузетно једноставна семантика. Она је праћена једноставном синтаксом *S*-израза. Читав језик почива на свега пет примитивних функција и предиката. То има за последицу да се интерпретатор за *Lisp* може написати на самом *Lisp*-у на свега једној страници.

У поређењу са неким савременим функционалним програмским језицима, као што су *ML* и *Haskell*, *Lisp* има већи број недостатка, од којих су најзначајнији: динамичка провера типова, немогућност дефинисања апстрактних типова података и релативно непрегледна синтакса.

---

[5] Обликовање програмског језика *Fortran* је започео *John Backus* 1953. године. Спецификација је била довршена 1954., прва корисничка документација 1956., а прва имплементација 1957. године. Читав пројекат је остварен у оквиру компаније *IBM*.

Програмски језик *Lisp* је до данас претрпео бројне трансформације. Познати савремени дијалекти *Lisp*-а су, пре свих, *Scheme* [Kels1998] и *Common Lisp* [Stee1993]. Програмски језик *Lisp* је осварио широк утицај на многе друге језике. Иако је највећи утицај у области функционалних програмских језика, има и више императивних језика у којима су неки елементи обликовани под утицајем *Lisp*-а, као што су *Python* [Ross2003] и *Ruby* [Mats2008].

1.3.2. ML

Програмски језик *ML* [Miln1997, Harp2005] је настао на Единбуршком универзитету током 70-их година XX века. Као руководилац тима и најутицајнији аутор обично се у први план истиче Робин Милнер.

ML је функционалан програмски језик са стриктном семантиком. Има веома широке могућности. У време када се појавио био је међу најпрактичнијим функционалним програмским језицима. Једноставна основа функционалног програмског језика је додавањем корисних особина других програмских језика унапређена у моћан програмски језик [Ghez1997].

Као и већина других савремених функционалних програмских језика, *ML* омогућава рад са функцијама вишег реда, полиморфним типовима података, уклапањем узорака, апстрактним типовима података и друго. Међутим, *ML* није чисто функционалан, због тога што омогућава извођење бочних ефеката. Подржан је и рад са изузецима.

Најзначајнија својства *ML*-а везана су за његов систем типова. Он је по први пут у програмирање увео строгу статичку проверу типова са аутоматским разрешавањем типова. На тај начин је значајно поједностављено дефинисање полиморфних функција и полиморфних структура података.

1.3.3. Haskell

Haskell [Hudak1992] је чисто функционалан програмски језик са лењом семантиком. Настао је као резултат обједињавања већег броја међусобно семантички сличних нестриктних функционалних програмских језика, а у циљу формирања шире заступљеног стандардног функционалног програмског језика. Постоји више имплементација програмског језика *Haskell*, како компилатора тако и интерпретатора.

Као и у случају *ML*-а, и карактеристике програмског језика *Haskell* обухватају већину особина савремених функционалних програмских језика. Подржан је рад са функцијама вишег реда и апстрактним и полиморфним типовима података. Провера типова се изводи статички, у фази превођења. Међу карактеристичне концепте *Haskell*-а спада имплементација монада, на којима почивају тзв. императивно функционално програмирање и чисто функционални улазно/излазни подсистем [Jones1993].

Haskell је један од најзаступљенијих функционалних програмских језика.

2. Релационе базе података

2.1. Системи за управљање базама података

Историја развоја рачунара је у великој мери повезана са растућом потребом да се поједностави руковање великим количинама података. За први систем за аутоматску обраду података сматра се машина за обраду бушених картица коју је развио Херман Холерит (*Herman Hollerith*), а која је 1890. године употребљавана за потребе пописа становништва у САД.

У првим годинама развоја дигиталних рачунара један од највећих проблема је представљала зависност начина примене рачунара (тј. развоја апликација) од конкретне хардверске основе. Почетком шездесетих година XX века направљени су значајни кораци у смеру одвајања програмера од хардвера дефинисањем и имплементирањем првих програмских језика. У то време настаје и појам *база података*, који је подразумевао информације похрањене у оквиру рачунарског система на конципиран и структуриран начин тако да се њима може управљати на начин који не зависи од конкретне машине.

Упоредо са развојем рачунарства и проширивањем примене рачунара, расла је и потреба за стандардизацијом језика за програмирање база података. Увођење магнетних дискова као новог уређаја за чување података донело је неке новине у односу на до тада примењиване магнетне траке. Док су се подаци похрањени на тракама могли обрађивати ефикасно само ако би обрада текла секвенцијално, примена дискова је омогућила ефикасан произвољан приступ подацима. Чарлс Бакман (*Charles Bachman*), у оквиру компаније *General Electrics*, представља 1961. године *IDS* (енгл. *integrate data store*, интегрисано чување података), који почива на мрежном моделу података [Bach1964].

Средином шездесетих година више произвођача рачунара су формирали Комитет за базе података (енгл. *Database Task Group – DBTG*) са циљем усмеравања даљег развоја база података. Ова група је 1971. године објавила формални стандард за управљање базама података, познат под називом *Codasyl*. Из ове групе је изостао тада највећи произвођач рачунара – *IBM*. *IBM* је 1968. године представио сопствени производ, *IMS*, који је делимично потекао из пројекта Аполо који је водила *NASA*. За разлику од *Codasyla*, *IMS* је заснован на хијерархијском моделу података [Meltz2005].

Мрежни и хијерархијски модели података се често називају навигацијским базама података јер су захтевали од корисника да непосредно управља приступањем подацима.

Чак је и Бакманов говор поводом добијања Тјурингове награде био насловљен "Програмер је навигатор".

На слабости оба понуђена модела података је указао Едгар Код (*Edgar (Ted) Codd*). Едгар Код се запослио у *IBM*-у 1949. године. Био је ангажован у истраживачким лабораторијама компаније, у Сан Хосеу (*San Jose*). Код је приметио да је основна слабост оба модела у очекивању да корисник (програмер, администратор) обави највећи део посла, док рачунарски системи само извршавају веома једноставне наредбе. Кроз низ техничких извештаја [Codd1969] и коначно 1970. у раду „Релациони модел података за велике дељене банке података“ [Codd1970], који се сматра једном од највећих прекретница у историји рачунарства, Едгар Код је представио потпуно нов начин организовања и управљања подацима.

Суштинска концептуална разлика у односу на све до тада познате моделе података је била у потпуном одвајању релационог модела података од уобичајених начина писања програма. Модели података, који су му претходили, били су обликовани са тежњом да се омогући што једноставнији програмски приступ подацима. За разлику од њих, релациони модел података је у потпуности посвећен самим подацима. Тако је омогућено да се подаци моделирају применом формалних математичких концепата, који су по својој природи статички. Код је на чврстим математичким основама припремио теоријску подлогу за развој релационих база података. Поред одвајања модела података од имплементације, друга значајна особина релационог модела је у томе што он програмеру омогућава да уместо појединачне обраде података, потребну операцију примени на читавом скупу података.

Због посвећености матичне компаније систему *IMS*, Код је имао релативно слабу подршку у сопственом радном окружењу. Ипак, релациони модел је имао непосредан и веома брз утицај на истраживаче. Почетком 1970-их година је започело више пројеката заснованих на релационом моделу. Међу најважније спадају *System R*, *Ingres*, релациони пројекат компаније *Honeywell* и *Oracle*.

Пројекат *System R* је започео почетком седамдесетих година, као прототип релационог система за управљање базама података. *IBM* је том пројекту у почетку придавао релативно скроман значај. Иако је представљао прототип, *System R* је примењиван и у пракси. Ране верзије су употребљаване на *MIT*-у, док су касније ушле у производна окружења аеронаутичке индустрије. Прва комерцијална верзија *IBM*-ових релационих система објављена је 1980. године под именом *SQL/DS*. Даљом надградњом настаје *DB2* који је објављен 1982. године. Друго издање система *DB2*, из 1985. године, представљало је прекретницу за *IBM*, јер је тада *IMS* потиснут у други план[6].

Пројекат *Ingres* је започео 1973. године на универзитету Беркли. У оквиру пројекта развоја система за географске податке, за потребе групе за економију, Мајкл Стонбрејкер (*Michael Stonebraker*) и Јуџин Вонг (*Eugene Wong*) су започели примену релационог модела података у пракси. У оквиру пројекта *Ingres* је дефинисан и упитни језик *QUEL*. Многи учесници пројекта *Ingres* наставили су касније своје ангажовање на развоју комерцијалних релационих система за управљање базама података у различитим компанијама.

[6] Иако се из више извора може доћи до информације да је *IMS* постепено угашен, то је далеко од истине. Верзије овог система се и даље одржавају. Иако се број корисника овог система не повећава значајно, чињеница је и да се не смањује.

Пројекат *Ingres* је окончан 1982. године, да би 1985. године био трансформисан у пројекат *Postgres* са циљем проширивања релационог модела објектним елементима. Пројекат је касније настављен од стране компаније *Ingres Corporation*, тако да је и данас активан као комерцијалан СУБП [Ingr:2009].

У јуну 1976. године *Honeywell* је објавио први комерцијални релациони систем за управљање базама података. Међутим, тај систем није почивао на *SQL*-у.

Историја упитног језика *SQL* је тесно везана за развој прототипа *System R*. Језик је настајао и постепено сазревао током неколико година као резултат рада истог тима. Како је *IBM* читавом пројекту придавао скроман значај, до 1979. године су сви резултати били јавно публиковани, укључујући и теоријске концепте и аспекте имплементације. То је допринело ширењу идеја о могућем комерцијалном успеху технологије и постепеном окупљању ширих кругова истраживача и програмера око *SQL*-а. Последица тога је била да је у већем броју других пројеката почела примена релационог модела података и *SQL*-а. Прва инсталација прототипа *System R* која је обухватала пуну функционалну имплементацију *SQL*-а изведена је 1977. године. Ипак, то још увек није било широко доступно решење.

Нешто касније, 1977. године, Ларри Елисон (*Lawrence J. Ellison*) оснива Лабораторију за развој софтвера (*Software Development Laboratory*) са циљем развоја релационог система за управљање базама података. Радни назив ове базе података је био *Oracle* [Ora:2002]. Захваљујући одлуци да се као упитни језик имплементира *SQL*, у основи компатибилан са упитним језиком прототипа *System R*, *Oracle* је понео титулу првог комерцијално расположивог релационог система за управљање базама података који подржава *SQL*. Прва верзија система *Oracle* носила је име *Oracle Version* 2 и објављена је 1979. године. Лабораторија за развој софтвера је касније прерасла у компанију *Oracle*.

До данас су релационе базе података постале уобичајено средство за старање о подацима. Више произвођача нуде своје релационе системе за управљање базама података у распону од персоналних рачунара до великих система. Најзаступљеније имплементације релационих система за управљање базама података су *IBM DB*2, *Oracle* и *Microsoft SQL Server* [Dela2009]. Постоји и већи број система за управљање базама података развијаних по моделу отвореног кода или у виду бесплатних јавних верзија, а који последњих година по својим могућностима, перформансама и поузданости полако достижу наведена комерцијална решења: *MySQL* [Lans2007], *SAP MaxDB* [SAP:2009], *Postgres* [Post:2007] и други.

Савремено схватање појма базе података обухвата и њену активну услужну димензију. База података је уређена колекција података која пружа услуге чувања, читања, претраживања и ажурирања података већем броју апликација и/или корисника. Рачунарски системи који омогућавају прављење база података и њихово функционисање називају се *системи за управљање базама података* (СУБП). СУБП који почива на релационом моделу података назива се *релациони систем за управљање базама података* (РСУБП) [Date2003] [Pavl1996].

Системи за управљање базама података представљају један од основних темеља савременог развоја софтвера. Док програмски језици омогућавају формулисање алгоритама и на тај начин служе као средство за прецизно записивање послова које рачунар мора урадити, дотле системи за управљање базама података омогућавају трајно чување података у организованом облику и управљање тако сачуваним подацима.

Системи за управљање базама података се уобичајено имплементирају као сервиси, тј. као апликације које пружају услуге другим апликацијама. Због тога је за њих прикладна архитектура клијент-сервер. Улогу сервера, који пружа услуге, има СУБП, док улогу клијента имају апликације које употребљавају услуге система. СУБП представља активног учесника, који се стара о подацима. Он чува, претражује, чита и одржава податке у складу са захтевима клијената и дефинисаном структуром базе података, што обухвата и правила о одржавању интегритета података.

У случају великих база података СУБП се често имплементира као дистрибуирана база података, уз распоређивање функција на више различитих, међусобно повезаних и усклађених рачунара. Рачунари који чине дистрибуирани систем могу делити само послове процесирања, само послове чувања података или и једне и друге послове. У зависности од архитектуре, за комуникацију са клијентима може бити задужен један (или више) специфичних рачунара у оквиру система, или комуникацију може да остварује сваки од рачунара који чине систем. У сваком случају, из угла клијента, СУБП и тада функционише као једна целина која му пружа одговарајуће услуге, тако да се из угла употребе систем и тада може посматрати у складу са архитектуром клијент-сервер.

2.2. Релациони модел података

Релациони модел података се састоји од структурног дела модела, који се односи на логичку организацију података, и од манипулативног дела модела, који се односи на теоријске основе рада са тако организованим подацима. Основу за структурни део релационог модела представљају математичке конструкције:

- домен и
- релација.

У контексту релационог модела података, *домен* је скуп неких вредности, са припадајућим операторима. Појам *домена* у релационом моделу је практично еквивалентан појму *типа података* у теорији програмских језика.

Релације представљају један од основних математичких појмова. У контексту релационог модела података математички појам релације се у одређеном смислу проширује, али без значајне промене саме суштине појма. Централна улога појма релације је одредила и назив читавог модела података.

У релационом моделу, *релација R* над неким доменима $D1$, $D2$,... Dn је подскуп производа домена:

$$R \subseteq D1 \times D2 \times \cdots Dn$$

Свака од димензија производа представља домен једног *атрибута релације*. Производ домена атрибута се назива *доменом релације*. Сваки атрибут се именује и при раду се уместо индекса (тј. редног броја) домена употребљавају називи атрибута. Штавише, индекси се потпуно занемарују, тако да се за релације које се разликују само по редоследу атрибута сматра да имају исти домен. Због тога је при описивању релације (у случајевима када се не разматра нити апстрахује конкретан садржај релације) довољно навести скуп свих атрибута, са њиховим појединачним називима и доменима. Такав опис релације се назива *схема* релације. *Релациона база података* је скуп релација. *Схема*

релационе базе података је скуп схема релација које чине ту базу података. Елементи релација се називају *торке*.

Релације се обично визуално представљају у виду табела, тако да колоне одговарају атрибутима а редови торкама. Због тога се у пракси примењује и алтернативна терминологија: *табела*, *колона*, *ред*, при чему ови термини одговарају редом терминима: *релација*, *атрибут*, *торка*.

2.3. Елементи логичке структуре

Савременим верзијама стандарда упитног језика *SQL* практично се прописују и стандарди за логичку архитектуру. Логичка структура базе података обухвата више различитих елемената, при чему сама архитектура и односи ових појмова нису идентични за све расположиве системе.

Као што је већ наведено, основни елементи логичке структуре су домени и релације. Док се концепт релације практично увек имплементира у складу са стандардом, дотле је концепт домена и рада са доменима обично имплементиран без веће сагласности са стандардом, у минималном обиму који је неопходан за имплементацију релација. Поред домена и релација, у најважније концепте логичке структуре релационих база података спадају и:

- База података – База података представља једну колекцију података о којој се стара РСУБП. Иако већина РСУБП омогућава да један систем може обухватати више различитих база података, у случају неких имплементација један РСУБП се стара о тачно једној бази података.
- Схема – У самом релационом моделу података, појам *схема релационе базе података* представља скуп схема релација које чине ту базу података. Међутим, због једноставнијег сналажења у већим базама података, које имају велики број релација, када се ради о имплементацијама РСУБП-а, појам схеме подразумева један именовани скуп релација. Једна база података може имати више схема. По стандарду, припадност релације некој схеми не имплицира никакве друге особине те релације. Међутим, у системима који подржавају тачно једну базу података, сама схема има далеко значајнију улогу и практично представља еквивалент стандардизованом концепту базе података. Неки системи уопште не подржавају схеме, већ их изједначавају са појмом *власништва* над релацијом (тј. уместо имена схеме користе име власника релације). При томе, појам власништва подразумева да *власник* релације (или неког другог објекта) има све привилегије у односу на ту релацију (или објекат).
- Кључ – Скуп атрибута једне релације који има посебан значај назива се *кључ*. У имплементацијама релационих база података од значаја је више врста кључева:
    - Јединствени кључ – Ако за кључ важи да релација не сме садржати два реда за које су вредности атрибута који чине кључ међусобно једнаки, такав кључ се назива *јединствени кључ*.
    - Кандидат за кључ – Ако за јединствени кључ важи да је то истовремено и минималан јединствени кључ, тј. да ниједан његов подскуп није јединствен кључ за дату релацију, онда се такав кључ назива *кандидат за кључ*.

- Примарни кључ – Један од кандидата за кључ се бира за примарни кључ. Обично се бира кључ који представља најприроднији и најчешћи критеријум за идентификовање торки неке релације. По стандарду *SQL*-а, свака релација у оперативној фази свог постојања мора имати тачно један примарни кључ.
- Страни кључ – Кључ једне релације који се употребљава за једнозначно референсање на торке друге релације назива се *страни кључ*. Страни кључ може да одговара било ком кључу кандидату референсане релације, а најчешће се односи управо на њен примарни кључ. Страни кључеви су основно средство за дефинисање правила референцијалног интегритета.

- Услов – Ако је неопходно да вредности атрибута неке релације буду међусобно усклађене по неком критеријуму, тада се на релацији дефинише *услов* (или *уведено ограничење*) који мора бити испуњен за атрибуте сваке појединачне торке. Услови нису саставни део релационог модела, али представљају стандардизовани елемент релационих база података.
- Привилегија – Обезбеђивање поузданости и безбедности података спада у примарне функције сваког система за управљање базама података. У области РСУБП-а се управљање безбедношћу остварује применом *привилегија*. Да би неки корисник могао да изврши неку операцију на неком објекту базе података или на неком конкретном податку, мора имати одговарајућу привилегију. Свака привилегија обухвата предмет на који се односи (објекат или податак базе података), субјекат коме се додељује и врсту и обим допуштених активности.
- Поглед – Поглед је елемемент презентационог слоја релационог модела података. Поглед се дефинише као скуп података који се динамички израчунава на основу садржаја релација и датог упита. Сам модел не ограничава значајно функционалност погледа, али се ради ефикасности и недвосмислености обављања одређених операција у имплементацијама обично ограничава функционалност погледа тако да се погледи примарно употребљавају за читање, док се само одређене врсте погледа могу употребљавати и за мењање података[7]. Неке савремене верзије РСУБП-а (нпр, *IBM DB2*, од верзије 7) омогућавају постављање окидача на погледе ради подржавања дословно свих операција одржавања података као у случају релација.
- Индекс – Индекси служе за обезбеђивање ефикаснијег приступа садржају релација. Иако они представљају део физичке организације података, наведени су у оквиру логичких елемената јер у неким имплементацијама представљају једино средство за остваривање јединствених кључева.
- Активни елементи – За разлику од свих претходних елемената логичке структуре базе података, који имају искључиво декларативну семантику, неки еле-

[7] Да би се поглед могао употребљавати за мењање података, упит којим се поглед дефинише обично мора да задовољи неколико услова: главни део упита мора да буде над тачно једном релацијом или погледом који допушта мењање; главни део упита не сме да садржи колонске функције; подупити морају да задовољавају неке услове који зависе од конкретне имплементације.

менти имају оперативну семантику. Основни активни елементи релационих база података су:

- Окидач – Већ је наведено да кључеви (јединствени, примарни, страни) и услови имају улогу у одржавању интегритета података. Поред њих, за старање о интегритету података се, као активни елементи, употребљавају *окидачи*. Окидач је елемент базе података који омогућава да се у одређеним случајевима аутоматски покрене извршавање неког програмираног поступка. Стандардизовани догађаји су додавање, мењање и брисање торки неке релације. Окидачи могу да се активирају пре или после активирајућег догађаја. Могу се активирати једанпут за читаву наредбу која их активира, или за сваку обухваћену торку посебно. Неки системи (нпр. *Oracle*) омогућавају постављање окидача не само на релације него и на схеме. Неки други системи (нпр. *IBM DB2*) омогућавају постављање окидача на погледе.
- Кориснички дефинисана функција – Већина РСУБП-а омогућавају писање функција (енгл. *user defined function – UDF*), које се затим могу употребљавати на исти начин као и уграђене функције *SQL*-а.
- Серверска процедура – Процедуре дефинисане на серверима (енгл. *stored procedure*) се употребљавају за централизовано дефинисање неких сложенијих поступака читања или одржавања података. На тај начин се предупређују евентуалне неусклађености међу апликацијама које употребљавају базу података и потенцијално се смањује обим протока података између РСУБП-а и апликације. Старије имплементације су имале уско специфичне начине писања процедура и интерфејсе према језицима који су се за то употребљавали. Савремене имплементације се нешто боље придржавају стандарда, који сада формално прописује не само интерфејсе према другим програмским језицима него и два основна начина за писање процедура:
  - процедурални део упитног језика *SQL* и
  - интерфејсе према програмском језику *Java* и начине њихове употребе.

2.4. Упитни језик *SQL*

Манипулативни део релационог модела дефинише формалне упитне језике и елементе модела који се тичу одржавања интегритета података. *Релациона алгебра* је формални упитни језик који представља надградњу уобичајене скуповне алгебре. Релациона алгебра има операциону семантику: упити представљају изразе које је потребно израчунати да би се добио тражени резултат. Са друге стране, *релациони рачун* се дефинише као надградња предикатског рачуна првог реда. Он има декларативну семантику: упити представљају логичке изразе које морају задовољити торке које чине резултат упита (и само оне).

Назив упитног језика *SQL* је изворно представљао скраћеницу за „Структурирани упитни језик“ (енгл. *Structured Query Language*), али се у актуелној верзији стандарда језика

назив *SQL* употребљава као конкретан назив, а не као скраћеница. Први стандард *SQL*-а је дефинисан од стране организација за стандарде *ISO* и *ANSI* 1986. године [ISO:1986]. За разлику од неких других стандарда у рачунарству, који формализују актуелно стање усвајањем пресека особина које су заједничке за већину најутицајнијих производа у области, стандард *SQL*-а је од самог почетка обликован тако да усмерава развој наредних верзија релационих СУБП-а. Та „далековидост" иде толико напред, да и 20 година након првог стандарда, још увек не постоји комерцијално расположив СУБП који је у потпуности у сагласности са тим стандардом.

Прва већа ревизија стандарда је довршена 1992. године. Званично се означава са *SQL*-92, али се неформално често помиње као *SQL*2. Донела је већи број мањих измена и допуна, као и неке сасвим нове концепте. Њен посебан значај је био у практичном поклапању објављивања са периодом велике експанзије *PC* сервера и великог пораста примене релационих система за управљање базама података. Ова верзија стандарда је покренула значајно приближавање постојећих имплементација стандардној верзији.

Наредна ревизија је објављена 1999. године и по свом обиму и значају је представљала изузетан резултат [ISO:1999]. Ова верзија се званично означава као *SQL*:1999, а незванично је у оптицају назив *SQL*3. Овом верзијом стандарда је обухваћено много више садржаја од самог упитног језика. Детаљно су покривени концепти и архитектура релационих база података, сам упитни језик, различити аспекти повезивања са програмским језицима, као и међусобног повезивања више релационих система. Основне новине у самом језику *SQL* представљали су елементи за подршку хијерархија типова и изградњу објектно оријентисаних проширења релационих база података. Упитни језик је проширен процедуралним елементима тако да од ове верзије стандарда и процедуре и функције које се извршавају у оквиру СУБП-а могу да се пишу на *SQL*-у.

Верзија стандарда *SQL*:2003 је наставила да проширује сопствени домен. Стандард је допуњен аспектима рада са документима у формату *XML*. Посебан том стандарда је посвећен употреби програмског језика *Java* за писање серверских процедура и функција. Унапређен је део који се односи на објектно оријентисане елементе упитног језика. Наредна верзија је *SQL*:2006. У њој је настављено са унапређењима у истом смеру. У упитни језик је интегрисан језик *XQuery* за руковање садржајем докумената у формату *XML*. Током истраживања и писања дисертације објављен је нови стандард *SQL*:2008 [ISO:2008].

И поред постојања стандарда језика, практично сваки РСУБП се одликује специфичним дијалектом *SQL*-а. Разлози постојања разлика у односу на стандард су углавном:

- недоследност у (не)придржавању стандардом дефинисаних концепата релационог модела;
- потреба да се одржи сагласност са претходним верзијама истог производа, које су настале пре доношења стандарда;
- политички разлози, као што је тежња неких произвођача да олакшају преношење решења развијених за неки нерелациони систем на њихов систем, или да отежају преношење развијених решења са њиховог на неки други РСУБП и слично.

И поред значајних проблема, мора се приметити да са сваком наредном верзијом, произвођачи приметно теже приближавању актуелном стандарду. У овом тексту ће се углавном употребљавати елементи језика који нису мењани од верзије *SQL*:1999, тако да би све изнесено у вези упитног језика *SQL* требало да буде у складу са стандардима *SQL*:1999, *SQL*:2003 и *SQL*:2006, а у највећој мери и са верзијом *SQL*-92. У неким случајевима, у којима одступање од стандарда буде неизбежно, користиће се дијалект упитног језика *SQL* система *IBM DB2*, верзија 9.5 [IBM:2008].

Актуелан стандард упитног језика *SQL* се састоји од неколико целина од којих су најважније: језик за управљање подацима (енгл. *data manipulation language*, *DML*), језик за дефинисање података (енгл. *data definition language*, *DDL*), језик за контролу података (енгл. *data control language*, *DCL*) и процедурални *SQL*. У наредним одељцима су укратко описани основни елементи језика за управљање подацима. Процедурални *SQL* ће бити представљен у одељку 5.3. Остале целине нису посебно значајне у контексту овог рада.

2.4.1. Упити

Основни квалитет упитног језика *SQL* је у апстраховању концепта упита на начин како је то предвиђено релационим моделом података. Манипулативни део модела дефинише упит као *израз над релацијама који дефинише (или израчунава) нову релацију*. Штавише, у *SQL*-у се уводи еквиваленција између једне скаларне вредности и релације која има тачно један атрибут и тачно једну торку (вредност јединог атрибута једине торке је употребљава као одговарајућа скаларна вредност), као и између коначног скупа скалара (истог типа) и релације која има тачно један атрибут (вредности тог атрибута у торкама релације се употребљавају као скуп скалара). На тај начин је омогућено да у самом упиту, а у складу са синтаксом језика:

- на сваком месту у упиту на коме се може појавити релација, уместо релације може да се употреби неки упит чији резултат одређује релацију одговарајућег типа (тј. домена);
- на сваком месту на коме се може појавити скуп скалара, уместо скупа скалара може да се употреби неки упит чији резултат одређује релацију са тачно једним атрибутом одговарајућег типа и
- на сваком месту на коме се може појавити скалар, уместо скалара може да се употреби неки упит чији резултат одређује релацију са тачно једним атрибутом и тачно једном торком одговарајућег типа.

Упит који се налази унутар већег упита назива се *подупит*.

Могућност коришћења мањих упита ради изградње сложених упита је обезбедила да се на *SQL*-у могу писати веома сложени упити на добро структуриран и прегледан начин, уз разбијање сложених проблема на мање и лакше решиве целине. Стандард упитног језика не прописује допуштену дубину подупита[8], али конкретне имплементације из техничких разлога уводе нека ограничења дубине и/или укупне дужине текста упита, која обично нису практично достижна.

[8] Дубина упита је број нивоа уњеждења упита у главном упиту. На пример, ако у упиту *У0* имамо подупит *У1*, а у њему подупит *У2*, па тако до *Уn* тада је дубина упита управо *n*.

Могућност структурирања упита је утицала и на избор иницијалног имена језика: *Структурирани упитни језик*.

Наредба SELECT

Основна наредба за израчунавање резултата упита у *SQL*-у је наредба `SELECT`. Упрошћени облик ове наредбе је:

```
SELECT <листа атрибута>
FROM   <спајање>
[ WHERE  <услов> ]
```

Чине је три дела (клаузуле):

- у делу `SELECT` се наводи листа атрибута које је потребно издвојити;
- у делу `FROM` се наводи спајање релација из којих се подаци издвајају;
- део `WHERE` садржи услов који се употребљава као критеријум издвајања торки из скупа свих торки добијених спајањем релација наведених у делу `FROM`.

Спајање се записује у облику:

```
<спајање> ::=
    <релација>
    | <спајање> <оператор спајања> <спајање>
```

Стандард прописује више различитих оператора спајања, од чега је само мањи број подржан у имплементацијама. Основне врсте спајања, које су подржане у свим значајнијим имплементацијама су *унутрашње спајање* (оператор `INNER JOIN` или само `JOIN`), *спољашње спајање* (`LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, `FULL OUTER JOIN`) и *производ* (`CROSS JOIN` или само запета: ‘`,`’).

Резултат упита је релација чији су схема и садржај одређени упитом. Семантика израчунавања упита одговара извршавању трију основних операција релационе алгебре: најпре се изводи спајање (или множење) свих релација, па рестрикција по датом услову и на крају пројекција на потребне атрибуте. У случају спајања, семантика упита се може записати као:

```
пројекција(
    рестрикција( <спајање>, <услов> ),
    <листа атрибута>
)
```

У случају множења семантика упита се може записати као:

```
пројекција(
    рестрикција(
        множење_релација( <листа релација> ),
        <услов>
    ),
    <листа атрибута>
)
```

Услов у клаузули `WHERE` представља логички израз који може бити компонован од више једноставнијих логичких израза уз примену логичких оператора `AND`, `OR` и `NOT`. Елементарни делови логичког израза, који не садрже логичке операторе и имају истинитосну вредност називају се *предикати*. Основни облици предиката су *предикати*

поређења који представљају изразе упоређивања вредности двају скаларних израза. Подржани су уобичајени оператори поређења који се могу употребљавати на нумеричким, знаковним и временским типовима података: =, <>, <, >, <= и >=. Поред тога постоји и већи број специфичних предиката. Ако је критеријум рестрикције универзалан (тј. потребно је издвојити све торке) део `WHERE` се може изоставити.

Релације и атрибути се у општем случају идентификују својим квалификованим именима, у облику <*име_схеме*>.<*име_релације*> и <*име_схеме*>.<*име_релације*> .<*име_атрибута*>. Име релације може бити неквалификовано (без имена схеме) само ако се релација налази у подразумеваној схеми. Име атрибута може бити неквалификовано (без имена релације и схеме) само ако атрибут са датим именом постоји у тачно једној од релација наведених у клаузули `FROM`.

Уколико се желе издвојити сви расположиви атрибути, онда се у клаузули `SELECT` уместо листе назива атрибута може навести симбол '*'.

```
select student.indeks, student.ime, student.prezime,
       ispit.dat_polaganja, ispit.ocena
from student join ispit
        on student.indeks = ispit.indeks
    join predmet
        on predmet.id = ispit.id_predmeta
where predmet.sifra = 'P101'
```

Пример 1: Упит који израчунава податке о студентима који су положили испит из предмета са шифром *P101*

Подупити

Подупити се могу уграђивати у главни упит на више различитих начина. Најважнији начини су: употреба подупита при спајању, употреба предиката `EXISTS` и скуповне операције.

Употреба подупита при спајању подразумева да се у делу `FROM` уместо имена релације наведе подупит. Подупит се обавезно наводи у заградама, након чега се обавезно уводи име којим се реферише релација која се израчунава као резултат подупита:

`FROM (` <*подупит*> **`)`** `[`**`AS`**`]` <*име*> `...`

Предикат `EXISTS` израчунава логичку вредност *тачно* акко је вредност подупита непразна релација. Представља, на одређени начин, еквивалент егзистенцијалном квантификатору предикатског рачуна првог реда. Употребљава се у облику:

`...` **`EXISTS(`** <*подупит*> **`)`**`...`

SQL подржава основне скуповне операције над подупитима. Да би се на два подупита могла применити скуповна операција неопходно је да подупити буду сагласног типа[9]. За већину примена потребно је и да одговарајући атрибути имају идентичне називе, али то се може превазићи експлицитним именовањем атрибута резултата. Скуповне операције уније, пресека и разлике се означавају, редом,

[9] У случају скуповних операција се узима у обзир редослед атрибута подупита, што није у складу са релационим моделом.

операторима UNION, INTERSECT и EXCEPT. Посебно важна особина скуповних операција је да се оне понашају доследно у складу са релационим моделом, као праве скуповне операције. Чак и када нека од релација операнада има поновљене торке (што *SQL* допушта, за разлику од релационог модела), резултат скуповне операције ће бити релација у којој нема поновљених торки. Постоје и верзије „скуповних операција" које узимају у обзир и поновљене редове: UNION ALL, INTERSECT ALL и EXCEPT ALL.

Скуповне операције се употребљавају у облику:

```
<подупит> <операција> <подупит>
```

Наредба WITH

Од верзије *SQL*:1999 стандард језика обухвата и наредбу WITH, која представља важно средство за писање сложених упита, укључујући чак и рекурзивне упите. Ова наредба представља природну надградњу употребе подупита у оквиру клаузуле FROM наредбе SELECT, обезбеђујући како чистију синтаксу, тако и далеко богатију семантику.

Наредба WITH има облик:

```
WITH <именован подупит> {, <именован подупит> }
<наредба SELECT>
```

Сваки именован подупит има облик:

```
<име> [( <листа назива атрибута> )] [AS] ( <подупит> )
```

При томе се у сваком подупиту могу употребљавати именовани подупити који му претходе, али и управо тај исти именовани подупит. Тако је помоћу наредбе WITH могуће писање рекурзивних подупита.

```
with prosek as (
    select indeks, avg(ocena+0.0) as ocena
    from ispit
    group by indeks
)
select *
from student join prosek
    on student.indeks = prosek.indeks
where prosek.ocena = ( select max(ocena) from prosek )
```

Пример 2: Упит који израчунава податке о студентима који имају највишу просечну оцену

2.4.2. Наредбе за мењање садржаја базе података

Наредбе за мењање података се заснивају на употреби клаузуле FROM за идентификовање торки које подлежу мењању. Три основне наредбе су:

- INSERT – додавање нових торки;
- UPDATE – мењање постојећих торки и
- DELETE – брисање постојећих торки.

Наредбе за мењање садржаја базе података су релативно једноставне. Потенцијална сложеност ових наредби почива у деловима наредби који представљају услове или упите.

Свака од наредби за мењање је атомична: или ће се извршити све измене описане наредбом или се неће променити ниједан податак у бази података.

Наредба INSERT

Наредба `INSERT` додаје датој релацији нове торке, које су резултат датог упита или су експлицитно наведене. У оба случаја торке које се додају морају имати исти број и тип атрибута као релација којој се торке додају. Наредба има облик:

```
INSERT INTO <релација> [ ( <листа назива атрибута> ) ]
<торке које се додају>
```

где се `<торке које се додају>` одређују као резултат упита или се експлицитно наводе као листа торки:

```
<торке које се додају> ::=
    <подупит> |
    VALUES ( <листа вредности> ) {, ( <листа вредности> )}
```

Наредба UPDATE

Наредба `UPDATE` мења вредности атрибута постојећих торки дате релације. Има облик:

```
UPDATE <релација>
SET <промене>
[ WHERE <услов> ]
```

где се промене наводе једна по једна или све заједно:

```
<промене> ::=
    <листа промена> |
    ( <листа атрибута> ) = ( <листа израза> | <подупит> )

<листа промена> ::= <атрибут>=<израз> {, <атрибут>=<израз>}
<листа атрибута> ::= <атрибут> {, <атрибут>}
<листа израза> ::= <израз> {, <израз>}
```

Један начин примене наредбе `UPDATE` подразумева навођење листе појединачних промена вредности атрибута, које се међусобно раздвајају запетама. Код другог начин мењања вредности најпре се наводи листа назива свих атрибута чије се вредности мењају, а затим листа израза чије вредности представљају нове вредности атрибута. Уместо листе израза може се навести и подупит. У том случају, резултат подупита мора бити релација која има одговарајући број атрибута, као и њихове типове и редослед. Штавише, резултат подупита мора бити релација са тачно једном торком.

У свим облицима наредбе `UPDATE` важи правило да се прво израчунавају све нове вредности, па се тек затим мењају вредност атрибута.

Услов клаузуле `WHERE` одређује које ће се торке дате релације мењати. Обликује се на исти начин као код упита. Услов може садржати подупите произвољне сложености, с тим да постоје одређена ограничења у односу на употребу саме релације чији се садржај мења у подупитима. Ако се клаузула `WHERE` изостави, наредба ће променити вредности свих торки релације.

Наредба DELETE

Наредба `DELETE` брише све торке дате релације које задовољавају дати услов. Има облик:

```
DELETE FROM <релација>
[ WHERE <услов> ]
```

Правила писања клаузуле `WHERE` су иста као у случају упита и наредбе `UPDATE`. Ако се клаузула `WHERE` изостави, наредба ће обрисати све торке релације.

2.4.3. Трансакције

Системи за управљање базама података морају да остварују већи број функција. При томе променљивост података и могућност да већи број корисника истовремено покушава да чита или мења исте податке представљају основне факторе сложености. Концепт *трансакције* представља основно средство за остваривање поузданости и интегритета података у СУБП-овима.

Појам трансакције одговара логичкој целини посла при раду са подацима која има следећа својства [Date2003]:

- *атомичност* – трансакција се увек извршава у целости или се не извршава ниједан њен део, тј. или ће све промене података обухваћене трансакцијом бити успешне, или ће све промене бити поништене;
- *конзистентност* – трансакција преводи једно конзистентно (исправно) стање података у друго;
- *изолованост* – последице извршавања једне трансакције не могу се препознати од стране других трансакција све до њеног завршетка;
- *трајност* – ефекти успешно изведене трансакције су трајни и могу се поништити само другом трансакцијом.

Свака операција приступања подацима или објектима базе података мора бити обухваћена неком трансакцијом. По стандарду *SQL*-а, трансакција започиње имплицитно, првим покушајем приступања неком податку или објекту базе података. Ток извођења операција које чине трансакцију је обавезно линеаран, тј. ако се започне једна операција трансакције, тада ниједна друга операција не сме да започне пре него што се започета операција не доврши. Трансакција се завршава потврђивањем или поништавањем. У случају потврђивања, све извршене промене се трајно записују у бази података и постају доступне свим корисницима базе података. У случају поништавања, све извршене промене се поништавају и неће бити доступне ниједном кориснику базе података. Препоручује се експлицитно завршавање трансакција, али различити системи омогућавају и имплицитно завршавање трансакција са различитим критеријумима имплицитног потврђивања или поништавања.

2.4.4. Одступања SQL-а од релационог модела

SQL је од самог настанка заснован на релационој алгебри и релационом рачуну. Може се рећи да су прве верзије језика имале семантику ближу алгебри, а синтаксу ближу рачуну. Савремене верзије су ближе релационом рачуну него релационој

алгебри. Ипак, као и савремени расположиви системи за управљање базама података, и *SQL* се одликује неким значајним одступањима од релационог модела.

Отклони од релационог модела су увођени појединачно, када се чинило да омогућавају једноставнију имплементацију неких посебних случајева, али је време показало да су донели више проблема него што се у почетку очекивало. Разлози за неке компромисе су били у потреби обезбеђивања вишег нивоа перформанси првих верзија релационих система, како би биле упоредиве са перформансама постојећих решења. Услед компатибилности, навика корисника и инертности произвођача такви компромиси су се задржали све до данас. Иако постоје истраживања таквих проблема, као и предлози смерова у којима би се могле одвијати модификације [Date2007], практично на тржишту још увек не постоје имплементације релационих система које доследно поштују релациони модел података.

Упитни језик *SQL* има више елемената који одступају од релационог модела. Неки од најважнијих су:

- Поновљене торке – Резултат упита може бити релација са више међусобно идентичних торки. Штавише, и обична релација може садржати поновљене торке ако није дефинисан примарни кључ.
- Неименовани атрибути – Атрибут резултата упита може бити неименован, што има за последицу да се такав атрибут не може референцирати у упитима.
- Поновљена имена атрибута – Резултат упита може садржати више атрибута са истим именом, што има за последицу да се такви атрибути не могу референцирати у упитима[10].
- Значај редоследа атрибута – Редослед дефинисања атрибута у *SQL*-у може бити значајан, на пример у случају скуповних операција или наредбе за додавање торки, што није у складу са релационим моделом података.
- Погледи без опције провере услова – На *SQL*-у је могуће направити поглед преко кога се могу додавати или мењати подаци тако да нове вредности не задовољавају услове погледа[11].
- Нису подржане релације без атрибута – Теоријски модел захтева постојање две релације без колона (једна празна и једна са тачно једном торком), што у *SQL*-у није подржано.
- Вредност *NULL* – Извесно најпроблематичније одступање је увођење недефинисане вредности *NULL*. Посебна вредност *NULL* је уведена у *SQL* како би се омогућило представљање различитих специјалних случајева, попут недостајућих, непознатих или недефинисаних вредности. Касније анализе су показале да се вредност *NULL* употребљава за означавање више од 20 различитих посебних случајева. Једна од непријатних последица њеног увођења је и тровалентна логика: резултат поређења које укључује вредност *NULL* не може да буде

[10] Конкретни системи обично аутоматски именују такве колоне, али се алгоритми додељивања имена међусобно разликују.

[11] Стандард *SQL*-а обухвата опциону клаузулу WITH CHECK OPTION, којом се обезбеђују провере услова погледа при мењању података. Међутим, тиме што је она опциона је омогућено прављење погледа који нису у складу са релационим моделом.

ни *тачно* ни *нетачно* већ мора бити логичка вредност *непознато*, што је такође једно од значења вредности *NULL*. Међутим, сам релациони модел почива на двовалентној логици и закону искључења трећег, па увођење вредности *NULL* има за последицу да је тако модификован и имплементиран модел заправо противречан.

3. Типови

3.1. Типизираност података, програма и језика

Међу најчешће узроке и последице грешака у програмирању спада примена неке операције на неодговарајући аргумент. Системи типова представљају средство да се та врста грешака препозна и отклони при писању и превођењу програма, и тиме отклони могућност њиховог појављивања у фази извршавања програма.

Тип неког податка (или неке променљиве) представља горње ограничење распона различитих вредности које тај податак може имати [Card2004][12]. У сваком програмском језику се може на концептуалном нивоу установити тзв. *тривијалан тип* као скуп свих могућих вредности са којима се може радити у програмима писаним на том програмском језику. За податак чији је домен ограничен неким (нетривијалним) типом каже се да је *типизиран*. За податак чији домен није ограничен (тј. ограничен је само тривијалним типом) каже се да је *нетипизиран*.

На одговарајући начин се, на основу типова аргумената и типова резултата, дефинишу и *типови потпрограма*.

Типизиран програмски језик је програмски језик у коме су сви подаци типизирани. Програмски језик у коме подаци нису типизирани је *нетипизиран програмски језик*. У неким типизираним програмским језицима постоје конструкције које омогућавају да се приликом извршавања програма на различите начине доведе у питање тип података са којима се ради. На пример, у програмском језику C се применом показивачке аритметике или експлицитним измењеним тумачењем типова (енгл. *cast*) могу начинити типовне грешке у фази извршавања програма. Такви језици се називају *слабо типизирани* језици или *језици са слабом провером типова*. Насупрот њима, програмски језици у којима такве грешке нису могуће су *строго типизирани језици* или *језици са строгом провером типова*.

Типизираност програмских језика је веома значајна особина, која има више значајних последица. Утицај типизираности се осећа највише у практичном раду и укључује:

[12] Појам типа и већина других појмова који се односе на типове а заступљени су у овом раду, наводе се и употребљавају у складу са одговарајућим дефиницијама уведеним у [Card2004].

- *Ефикасније извршавање програма.* Увођењем провере типова при превођењу програма може се обезбедити да се при извршавању програма, без додатних провера, може увек употребити најбоља имплементација одговарајуће операције. На пример, имплементације рачунских операција се разликују у зависности од типова података.
- *Економичност развоја у малим размерама.* Добро заснован систем типова омогућава да се у развоју сваке појединачне компоненте софтвера значајан део могућих грешака открије и отклони у раним фазама развоја. Штавише, и у случајевима када је неопходно дебаговати програм, ради проналажења и отклањања других врста грешака, типизираност доприноси скраћењу и поједностављењу тог поступка јер се велики број евентуалних грешака везаних за типове може одмах одбацити.
- *Економичност развоја у великим размерама.* Информације о типовима елемената програмских модула који се могу користити од стране других модула чине *интерфејс модула*. Дефинисањем интерфејса модула и пре саме имплементације модула омогућава се плански и координисан развој делова великих програмских система по модулима, у више мањих развојних тимова.
- *Безбедније и ефикасније повезивање модула.* Ако се за сваки модул посебно воде информације о типовима у том модулу, онда је могуће да се модул преведе једнократно, а да се више пута повезује при превођењу програма у којима се употребљава. Без информација о типовима, такво повезивање не би било безбедно.
- *Повећавање безбедности кода.* Велики број безбедносних пропуста у програмима је последица различитих слабости у области провере типова. Типизираност сужава простор за настајање и евентуално злонамерно искоришћавање таквих пропуста.

Са друге стране, типизираност на одређени начин умањује изражајну моћ програмског језика. На језицима који нису типизирани могу да се напишу неки потпрограми који се не могу написати на типизираним програмским језицима. На пример, на типизираним програмским језицима није могуће написати комбинатор Y, који је уобичајен у λ-рачуну:

$$Y = \lambda g.(\lambda x.g\,(x\,x))\,(\lambda x.g\,(x\,x))$$

Да би се комбинатор Y могао написати на типизираном програмском језику, неопходно је да одговарајући систем типова у потпуности подржава рекурзивне функцијске типове[13], што по правилу није задовољено. Основни разлог за изостављање рекурзивних функцијских типова из система типова програмских језика је у томе што они, иако имају непорецив теоријски значај у домену истраживања, у пракси доносе више штете него користи. Велики број грешака које се односе на типове се открива тако што се „препознају“ недопуштени рекурзивни типови, па би пуна подршка за рекурзивне типове онемогућила недвосмислено препознавање таквих грешака. Што се тиче евентуалних користи, испоставља се да све оно што може да се имплементира

[13] Рекурзиван функцијски тип је тип функције која се може применити на себе саму.

применом оператора са рекурзивним типовима, може да се имплементира и другачије, применом уобичајених метода рекурзије.

3.2. Системи типова

Провера исправности употребе типова у програмима се може обављати *статички* и *динамички*. Статичка провера типова се обавља у фази превођења програма и подразумева скуп различитих провера ради препознавања евентуалних грешака које се односе на типове. Динамичка провера типова се обавља у фази извршавања програма. Програмски језици у којима се користи статичка провера типова називају се *статички типизирани језици*. *Динамички типизирани језици* се одликују применом динамичке провере типова.

Провера типова почива на информацијама о:

- систему типова конкретног програмског језика;
- типовима уграђених елемената програмског језика и
- типовима елемената програма који се дефинишу и употребљавају у програму.

Систем типова је скуп логичких правила о извођењу закључака о типовима. Систем типова мора бити:

- одлучиво проверив – мора постојати *алгоритам провере типова*, који једнозначно установљава да ли је неки програм добро заснован;
- траспарентан – скуп правила мора бити такав да програмер сам, само на основу текста програма, може проверити тип неког израза и
- доследан – потребно је да правила система типова омогућавају што темељнију проверу типова не само на нивоу појединачних модула него и сложених програмских целина.

Доследност и одлучивост система типова се проверавају формалним заснивањем и представљањем система типова. Формални системи типова почивају на *формалној теорији типова* [Thom1999]. Док се у техничкој документацији (развојној и корисничкој) за програмске језике обично наводе неформални системи типова, формални системи типова представљају њихов формалан математички запис. Значај формалног заснивања система типова је у томе да се над формалним системом типова могу формалним путем проверавати одређена важна својства система типова, као што је важење теореме о сагласности типова [Miln1978], која тврди да ће добро засновани програми (тј. програми у којима су сви типови међусобно усаглашени) имати добро понашање, односно неће при извршавању производити грешке које се односе на типове.

Поступак проверавања исправности типова се имплементира као провера испуњености правила која чине систем типова. Претпоставке извођења се изводе из текста програма и декларација употребљаваних библиотека:

- на основу сваке експлицитне декларација типа неког имена установљава се претпоставка о познатом типу који одговара том имену;

- на основу сваке декларације употребљаваног елемента неке библиотеке[14] установљава се претпоставка о типу тог елемента и
- на основу сваке појединачне употреба неке операције или функције установљавају се претпоставке да је тип сваког појединачног израза, који представља аргумент операције, у сагласности са формалним типом аргумента те операције.

Правила која чине систем типова могу да у поступку проверавања типова имају и улогу правила извођења (или распознавања) типова. Алгоритам за проверавање типова има сличности са алгоритмима за формално доказивање теорема. Захваљујући томе што су скупови претпоставки и правила извођења у значајној мери специфичног карактера, алгоритми за проверавање типова се могу имплементирати као специфични случајеви општијих доказивача.

Резултат проверавања исправности типова је закључак да су све декларације и употребе типова међусобно усаглашене, или закључак да постоје неусаглашености. За програм у коме су сви типови међусобно усаглашени каже се да је *исправно типизиран*.

3.2.1. Типови у програмским језицима

У програмским језицима се проблему типова може приступати на више различитих начина. Као што је већ указано, поступак проверавања типова може бити статички или динамички. Друга значајна разлика је у начину типизирања дефинисаних имена. У већини савремених програмских језика се при дефинисању неког елемента програма експлицитно назначава и његов тип. Такви програмски језици су *експлицитно типизирани*.

Са друге стране, у неким језицима је дефинисањем формалнијег система типова и применом сложенијег алгоритма за проверавање типова омогућено писање програма без експлицитног назначавања типова. Такви језици су *имплицитно типизирани*. У имплицитно типизираним програмским језицима алгоритам за проверавање типова истовремено са проверама усаглашености изводи и закључке о типовима свих појединачних елемената програма.

Имплицитно типизирање има и предности и мане у односу на експлицитно типизирање. Изостављањем експлицитних декларација типова се скраћују записи програма, али се истовремено губи одређена документациона улога ових декларација. Имплицитним типизирањем се омогућава природно развијање полиморфних делова кода, али се отежава локализација грешака откривених при проверавању типова. Док се код експлицитно типизираног кода узроци евентуално уочене неусаглашености релативно лако проналазе у коду, у оквиру израза у којима до ње долази, дотле се код имплицитно типизираног кода такве грешке потенцијално одражавају на сасвим раздвојене делове кода.

У савременим програмским језицима се могу пронаћи и комбиновани системи типова. На пример, програмски језици *ML* и *Haskell* у неким деловима кода захтевају експлицитно типизирање, док у неким другим омогућавају имплицитно типизирање.

---

[14] Под елементом библиотеке се подразумева сваки податак, операција или било које друго име дефинисано у библиотеци.

3.2.2. Полиморфизам

Једна од значајних особина савремених система типова је подршка за полиморфне делове кода. За део кода се каже да је *полиморфан* ако у различитим контекстима може имати различите типове [Miln1978, Card1987].

Практично сваки програмски језик садржи неке обрисе полиморфизма, додуше у сасвим примитивном облику. Један пример је оператор сабирања, који у већини програмских језика може да се примени како на целе бројеве тако и на бројеве у запису са покретном запетом.

Ипак, појам полиморфизма подразумева далеко сложеније аспекте примењивости него што је то случај са операцијом сабирања. Основна карактеристика језика са *полиморфним системом типова* јесте да полиморфизам у њима није ограничен само на уграђене операције, већ да програмер може правити сопствене полиморфне делове кода.

Концепт полиморфности се у основи односи на операције (функције, процедуре), али се посредно преноси и на типове података, када се од полиморфних операција сачини један целовит модул који описује и структуру података. Већи број савремених програмских језика омогућава писање сложених структура података које су полиморфне у односу на неке своје компоненте. Међу најзначајније примере спадају колекције података засноване на шаблонима у програмском језику *C++*[15]. Ипак, најдаље су у том концепту отишли савремени функционални програмски језици, као *ML* и *Haskell*.

Данас се разликују три основне врсте полиморфизма:

- хијерархијски полиморфизам;
- параметарски полиморфизам и
- имплицитни полиморфизам.

Хијерархијски полиморфизам представља један од основних концепата објектно оријентисаних програмских језика. Основа хијерархијског полиморфизма је правило да све операције које се дефинишу за неки тип могу да се примене и на сваки други тип који представља његову специјализацију. У објектно оријентисаним програмским језицима се при разматрању хијерархијског полиморфизма као основа употребљава појам класе, који се односи само на један подскуп типова. У складу са тим, појму специјализације типова у објектно оријентисаним програмским језицима одговара појам наслеђивања класа. Одатле непосредно следи да је концепт хијерархијског полиморфизма општији него верзије тог концепта примењене у већини објектно оријентисаних програмских језика.

Параметарски полиморфизам почива на увођењу *типовних параметара*. У декларацијама и имплементацијама полиморфних операција и структура се употребљавају формални типовни параметри. При употреби полиморфних операција и структура се експлицитно наводе стварни типови[16].

---

[15] Одговарајуће конструкције постоје и у новијим верзијама програмских језика *Java* и *C#*.

[16] Под одређеним условима се стварни типовни параметри не морају наводити већ се аутоматски установљавају. На пример, при употреби шаблона функције у програмском језику

Имплицитни полиморфизам представља уопштење параметарског полиморфизма у смислу изостављања експлицитног декларисања типовних параметара. Имплицитни полиморфизам је тесно повезан са имплицитним означавањем типова и потпуно је остварив само у имплицитно типизираним програмским језицима.

Већи број програмских језика омогућава више врста полиморфизма. На пример, неки савремени објектно оријентисани програмски језици подржавају хијерархијски и параметарски полиморфизам. Већина типизираних функционалних програмских језика подржава параметарски или имплицитни, а неки подржавају и хијерархијски полиморфизам. Штавише, хијерархијски полиморфизам је често подржан у општијем облику него у императивним објектно оријентисаним програмским језицима.

У литератури не постоји сагласност око тога да ли су динамички типизирани језици полиморфни или не. Један број аутора им придаје полиморфност, али неки то доводе у питање[17]. У контексту овог рада је сматрано да полиморфност може бити особина само строго типизираних језика.

3.3. Распознавање типова

Једна од најважнијих фаза превођења програма написаних на неком имплицитно типизираном програмском језику је распознавање типова. Основни циљеви распознавања типова су:

- распознавање типова дефинисаних имена;
- установљавање типова које у конкретним изразима имају имена чије су дефиниције полиморфне и
- проверавање усаглашености типова у програму.

По својој природи проблем распознавања типова је веома сличан проблему проналажења модела који задовољава скуп хипотеза исказног рачуна. Улогу исказа овде имају хипотезе о типовима, а улогу променљивих имају типовне променљиве. У пракси се за распознавања типова најчешће примењује *Хиндли-Милнеров* алгоритам [Miln1978]. Хиндли-Милнеров алгоритам почива на Робинсоновом алгоритму унификације [Robi1965]. Алгоритам је касније детаљније разрађиван и унапређиван у бројним радовима [Dama1982], [Card1985], [Card1987] и [Card1991]. Овај алгоритам се помиње у литератури и као „Дамас-Милнеров алгоритам“ или „алгоритам *W*“.

Због специфичности конкретних програмских језика, имплементације овог алгоритма обично уводе одређене модификације. Једна од мана Хиндли-Милнеровог алгоритма је што у случају неуспеха (тј. у случају постојања типовних грешака у

C++, уколико се вредност (тј. тип) типовног параметра може једнозначно установити на основу наведеног аргумента функције, онда тај тип није неопходно експлицитно наводити.

[17] На пример, ако би на динамички типизираном језику била написана функција која израчунава обрнуту листу, та функција би била полиморфна у смислу да се „може применити на листу са произвољним типом елемената“, али не би била полиморфна у смислу да јој „одговара скуп типова“. Тренутак распознавања њеног типа (у тренутку израчунавања) имплицира да она до израчунавања нема никакав тип (није типизирана) а при израчунавању добија тачно један конкретан неполиморфан тип.

програму) не даје довољно тачну локацију узрока. Због тога је развојено неколико надградњи овог алгоритма, као и покушаја уопштавања различитих варијанти алгоритма [Heer2002].

Имплементација алгоритма распознавања типова за програмски језик *Wafl* је описана у одељку 4.3. Пример примене алгоритма је наведен у додатку, на страни 193.

4. Програмски језик *Wafl*

4.1. Основне особине *Wafl*-а

Програмски језик *Wafl* је обликован у оквиру истраживања примењивости функционалних програмских језика у области развоја Веб апликација [Малк2002]. Чињеница да је програмски језик *Wafl* у пуној мери обликован и имплементиран „сопственим снагама“, чини га посебно погодним за различите врсте експеримената у области програмских језика и њихових примена. У овом конкретном случају, због тога што је истраживање везано за типове података, посебно је важнан начин имплементације типова у *Wafl*-у. Због тога је било сасвим природно да *Wafl* буде изабран као програмски језик који би послужио као платформа за истраживања обухваћена овим радом.

Програмски језик *Wafl* је обликован тако да буде у што већој мери опште примењив. Иако се језик одликује већим бројем специфичности које се односе на развој Веб апликација, оне су уклопљене у језик тако да не умање његову општост. *Wafl* је концептуално сличан савременим функционалним језицима, као што су *Haskell* и *ML*. Програми на програмском језику *Wafl* су семантички еквивалентни изразима. Из практичних разлога, синтакса језика је обликована по узору на широко распрострањене програмске језике *C* и *C++*, мада је у неким елементима блиска синтакси програмског језика *ML*.

Wafl омогућава чисто функционално програмирање. Ипак, ради проширивања домена примењивости језика, допуњен је неким елементима који омогућавају бочне ефекте, па језик као целина није чисто функционалан.

4.1.1. Типови

Wafl је строго типизиран програмски језик са имплицитном статичком провером типова. Систем типова у *Wafl*-у почива на имплицитном полиморфизму. Статичка провера типова са аутоматским извођењем типова је задужена за препознавање и проверу типа сваког појединачног израза и програма као целине. Све провере типова се одвијају у фази анализе и превођења програма, пре него што започне његово израчунавање.

Прости типови података у *Wafl*-у су: *Int*, *Float*, *Bool* и *String*. Ниске (тип *String*) се употребљавају како за ниске знакова, тако и за ниске бајтова. Ниске бајтова могу да

представљају различите бинарне садржаје, као што су слике и други бинарни ресурси. Подржан је уобичајен скуп основних операција и функција над простим типовима, укључујући и експлицитне конверзије примитивних типова. Због доследне строге типизираности, у *Wafl*-у нису подржане имплицитне конверзије типова.

За означавање полиморфних типова се употребљавају типовне променљиве, у ознаци *'n*, где је *n* неки природан број.

Постоје три врсте комбинованих типова, који представљају полиморфне типове код којих типовна променљива има за домен скуп простих типова. Тип *Prime*[*'n*] представља неки од простих типова. Тип *Value*[*'n*] представља неки од простих типова осим типа *Bool*. Тип *Numeric*[*'n*] представља неки од нумеричких типова.

Три основне врсте колекција су: листе, низови и каталози. Оне су полиморфне, па могу садржати елементе било којих типова. Међутим, због доследне строге типизираности, сви елементи неке конкретне колекције морају бити истог типа. Ознаке ових типова су, редом: *List*[*T*], *Array*[*T*] и *Map*[*K*][*T*], где *T* представља тип елемента колекције, а *K* тип кључа по коме се приступа елементима каталога.

Поред колекција, у сложене типове спадају и торке и слогови. Торке и слогови су структурни типови. Торке се састоје од елемената којима се приступа по редном броју. За разлику од низова торке могу имати елементе различитих типова. Ознака типа торке је *Tuple*[*T1,...,Tn*], где типови *T*i представљају типове елемената торке. Слогови се састоје од атрибута (елемената) којима се приступа по имену. Ознака типа слога је *Record*[*attr1*:*T1,...,attr*n:*Tn*], где су *attr*i називи, а *T*i типови атрибута. Редослед атрибута у слогу није значајан.

Поред *тачних* типова торки и слогова, често се употребљавају и *непотпуни* типови торки и слогова. Када се експлицитно праве торке и слогови, онда је тип резултата таквог израза одговарајући тачан тип торке или слога. Међутим, када се торке и слогови употребљавају у неком изразу (или функцији), онда аргумент може бити и неког конкретнијег типа, јер може имати и друге елементе осим оних који су неопходни да се може употребити у телу дефиниције. У таквим случајевима се употребљавају непотпуни типови торки и слогова.

Непотпуни тип торке *Tuple*[*T1,...,Tn*][*'m*] је полиморфан тип торке, који има најмање *n* елемената датих типова, али можда има и неке друге елементе. Потенцијални додатни елементи представљају проширење торке, облика *'m=Tuple*[*Tn+1,...,Tn+k*].

Слично, тип *Record*[*attr1:T1,...,attrn:Tn*][*'m*] је полиморфан тип слога, који има најмање *n* атрибута са датим именом и типом, али можда има и неке друге атрибуте. Потенцијални додатни атрибути се представљају проширењем слога, облика *'m=Record*[*attrn+1:Tn+1,...,attrn+k:Tn+k*].

Функцијски типови се записују у облику (*T1**...**Tn->TR*), где типови *Ti* представљају типове аргумената функције, а тип *TR* је тип резултата. Тип функције без аргумената се записује као: (->*TR*).

4.1.2. Функције

У програмском језику *Wafl* се примена функција наводи уз употребу заграда. Аргументи се међусобно раздвајају запетама. Ради повећања читљивости вишеструке узастопне примене функција, која је веома честа у функционалим програмским језицима, у

Wafl-у је подржана и такозвана „синтакса стрелице“. Она је слична примени метода у објектно оријентисаним језицима: уместо да се функција примењује на неколико аргумената, може да се примени као „метод“ првог аргумента, док се преостали аргументи наводе у заградама. Наредни пример илуструје синтаксу стрелице кроз два пара еквивалентних израза, у којима се примена функција представља како уобичајеном синтаксом, тако и применом синтаксе стрелице:

```
strLen( subStr( s, 10, 5 ) )
    = s->subStr( 10, 5 )->strLen()

addPageFrame(buildGraph(Form(),”summary”),50,100))
    = Form()->buildGraph(”summary”)->addPageFrame(50,100)
```

Обе синтаксе могу бити заступљене у истом изразу, без ограничења. Синтакса стрелице је посебно корисна при програмирању обраде података или конструкције ресурса (докумената, Веб страница и сл.), где се кроз више корака долази до коначног резултата.

Wafl у потпуности подржава функције вишег реда и делимично израчунавање функција. Функције се могу употребљавати као аргументи или израчунавати као резултати функција. Могу се појављивати и као елементи структура или колекција.

Делимично израчунавање у *Wafl*-у није ограничено редоследом аргумената, као што је случај у већини функционалних програмских језика. Ненаведени аргументи се означавају доњом цртицом. У наредном примеру је делимично израчунавање у *Wafl*-у илустровано са два низа међусобно еквивалентних израза:

```
f(a,b,c,d) = f(_,b,_,d)(a,c) = f(_,b,_,_)(_,c,_)(a,_)(d)
map(l,f) = map(_,f)(l) = map(l,_)(f)
```

Подржани су неименовани функцијски изрази, који се уобичајено називају *ламбда функције* или *ламбда изрази*. Слично као у другим функционалним програмским језицима, ламбда изрази представљају средство за једноставно дефинисање неименованих функција које се једнократно употребљавају. Основна синтакса дефиниције ламбда функције има облик:

```
\<формални_параметри>:<израз>
```

На пример:

```
\x,y: sin(x)*y
```

Специфичност програмског језика *Wafl* је да ламбда изрази могу да се напишу тако да поред непосредно везаних параметара (тј. аргумената записане функције) обухвате и нека имена која су везана из контекста у коме се ламда израз налази. Пуна синтакса ламбда израза у програмском језику *Wafl* је:

```
\<формални_параметри> [# <листа_везаних_имена>]: <израз>
```

Формални параметри представљају аргументе функције која је овако записана. Листа везаних имена декларише списак имена чије вредности су познате у контексту у коме се израчунава ламбда израз, чиме се наглашава да се та имена са управо тим вредностима користисте у телу ламбда израза. Једно исто име се не сме појавити и као формалан параметар и као везано име. Ако се са `x1`,...,`xn` означи низ од `n` аргумената, са `c1`,...,`ck` низ од `k` имена доступних у контексту у коме се наводи ламбда израз, и са `_`,...,`_` низ од `n`

изостављених аргумената делимичног израчунавања, тада су наредни изрази међусобно еквивалентни:

```
\x1,...,xn # c1,...,ck: <izraz>
(\x1,...,xn,c1,...,ck: <izraz>)(_,...,_,c1,...,ck)
```

Примену ламбда израза илуструје наредна дефиниција функције `map2` која израчунава листу резултата примене дате функције `f` на све парове елемената датих листа `l1` и `l2` (Пример 3).

```
map2b(l1,l2,f) =
    l1->map( \x # f,l2: l2->map(f(x,_)) )
      ->appendLists();
```

Пример 3: Пример ламбда израза у програмском језику *Wafl*

4.1.3. Специфичности које се односе на Веб

Wafl омогућава да се при писању веб апликација примењује било који од три основна приступа: програмски, шаблонски и сесијски. Тиме се издваја од свих других решења за развој Веб апликација познатих аутору. Штавише, могуће је и комбиновање различитих приступа у истом програму.

У програмима се могу употребљавати подаци специфични за конкретну сесију. О праћењу тока сесије и међусобном изоловању података сваке од сесија се стара окружење. Програмер може експлицитно управљати током сесије помоћу дијалога.

Wafl је прилагођен аутоматском обликовању докумената у форматима *HTML*, *XML* или *SGML*, као и других докумената који у основи имају текстуални запис. Сама синтакса текстуалних шаблона дефинисана је по узору на *HTML* али је једнако погодна и за *SGML*, *XML*, *TeX*, *RTF* и друге текстуалне формате докумената.

Висок ниво модуларности графичког дизајна је постигнут кроз концепте текстуалних шаблона докумената и библиотека текстуалних шаблона. Увођењем текстуалних шаблона је омогућена употреба независних алата за прављење шаблона докумената и једноставно укључивање аутоматски генерисаних делова страница у такве шаблоне.

Омогућавањем модуларног развоја корисничког интерфејса истовремено је омогућено и раздвајање корисничког интерфејса од логике апликације. Посебно средство за издвојено дефинисање сервисне логике је механизам Веб дијалога. Ове могућности раздвајања програмских целина ни на који начин не обавезују програмера, већ он сам може према конкретним потребама одлучити да ли му је и које раздвајање потребно.

Веб апликација развијана на *Wafl*-у се назива „*Wafl* услуга“. Једна *Wafl* услуга се састоји од већег броја појединачних програма. Сви програми који чине услугу имају приступ дељеној колекцији података услуге и различитим инстанцама дељених колекција података сесије. Док се подаци услуге могу програмски само читати (постављају се конфигурисањем услуге), дотле се подаци сесије могу и читати и мењати. Сви програми које покрене (имплицитно или експлицитно) један корисник током једне сесије употребе услуге, користе исту инстанцу података сесије.

4.2. Имплементација програмског језика

Ради разумевања неких специфичности окружења у коме је рађено истраживање, у наредним одељцима су укратко описани основни концепти на којима почива употребљавана имплементација програмског језика *Wafl*.

Сама дефиниција програмског језика *Wafl* оставља могућност да се програмски језик имплементира како у облику интерпретатора тако и у облику преводиоца. Ипак, због тога што је иницијално примарна намена *Wafl*-а била развијање Веб апликација, при имплементирању је било погодније одлучити се за интерпретацију. Више је разлога за такав избор. Један од најважнијих је што интерпретирање пружа краћи развојни циклус, који је од посебног значаја при развоју Веб апликација. Други разлог је што уобичајена предност компилирања у погледу ефикасности није посебно значајна у домену Веба, јер се програми обично довољно једноставни да повећана ефикасност извршавања није посебно значајна. Поред тога, извршавање великог броја малих компилираних програма умањује перформансе због повећаног ангажовања меморијских и процесорских ресурса при учесталом покретању и затварању процеса, док се једна иста инстанца интерпретатора употребљава за извршавање више инстанци програма.

При имплементирању интерпретатора за *Wafl* коришћена су позната искуства представљена у научној и стручној литератури, а посебно [Turn1979], [Hend1980], [Jones1987], [Cous1987], [Jones1992] и [Moun1999].

4.2.1. Превођење програма

Иако се у случају интерпретирања програма поступци превођења и извршавања програма могу преплитати, у случају имплементације интерпретатора за *Wafl* такво преплитање не постоји.

Први корак који се предузима је *синтаксна анализа*. Она обухвата токенизацију и парсирање текста програма или дела програма. Резултат синтаксне анализе је привремена репрезентација програма или дела програма у меморији. Она има облик *неповезаног графа*. Овај граф је *неповезан* зато што у овој фази превођења свака целина програма (тј. функција или сам програм) постоји као засебан граф који није повезан са графовима који одговарају функцијама које се у тој целини употребљавају.

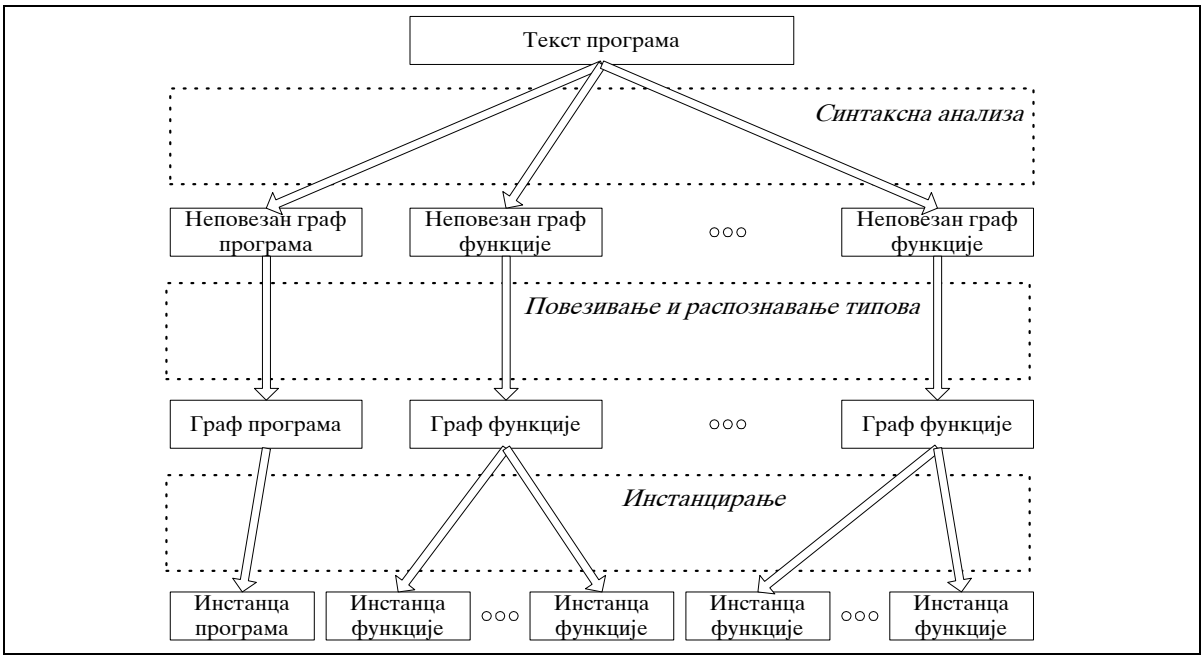
Други корак представља *повезивање графова* програма и функција. У овој фази се повезују референце на имена са дефиницијама које одговарају тим именима. У неким случајевима једном имену може одговарати више дефиниција. У овој фази се за сваку конкретну употребу имена формира скуп свих могућих дефиниција које му одговарају. Сужавање се оставља за наредни корак.

Трећи корак је *распознавање типова*. Аутоматско распознавање типова представља облик семантичке анализе програма. Распознавањем типова се сужавају скупови дефиниција које одговарају употреби неког имена, а према контексту у коме се име употребљава, тј. према типу објекта (или функције) на који се тим именом реферише. Резултат ове фазе су графовима представљени изрази који чине потенцијално полиморфне дефиниције функција и програма.

Повезивање графова и распознавање типова су тесно везани, па се може рећи да чине једну сложену фазу у превођењу програма. У случају имена са више значења (поли-

морфне дефиниције или функције које имају различите дефиниције за различите типове), при проверавању типова се установљава које се од значења имена може применити на ком месту.

Последњи корак у оквиру превођења је *инстанцирање кода*. Да би се програм могао извршити потребно је да се граф његове дефиниције преведе у облик погодан за израчунавање, тј. у тзв. *израчунљиву форму*[18]. У израчунљивој форми израза је тачно одређен тип свих операција и аргумената. У њој нема никаквог полиморфизма. Поступак превођења дефиниција функција и програма из облика потенцијално полиморфног графа у неполиморфну израчунљиву форму назива се инстанцирање. Инстанцирање почива на алгоритмима за редукцију типова који су веома слични алгоритмима за распознавање типова. Усвојени назив је последица чињенице да се полиморфне функције могу преводити у више различитих инстанци – по једна посебна инстанца за сваки конкретан тип са којим се полиморфна функција употребљава.



Слика 1: Фазе превођења програма

Наведени кораци чине *превођење* програма (Слика 1). Добром имплементацијом саме организације превођења и применом одговарајућих оптимизација могу се остварити значајни добици у перформансама самог превођења, али и обезбедити претпоставке за ефикасније израчунавање програма. Анализа могућих оптимизација се може обављати већ у фази анализе неповезаног графа, али се саме оптимизације најефикасније могу укључити у завршне фазе превођења, а пре свега у инстанцирање кода.

По обављеном превођењу може се приступити *израчунавању* програма. Инстанциран програм се састоји од израчунљиве форме израза којим је програм дефинисан, а у којој се употребљавају и израчунљиве форме израза којима су дефинисане функције. Такав програм се може израчунавати у оквиру извршног окружења *Wafl* интерпретатора.

[18] Појам *израчунљива форма израза* означава форму коју израз добија по окончању превођења. Назива се *израчунљивом* јер јер то форма израза која на крају учествује у израчунавању. Овај појам нема никакве везе са појмом израчунљивости у теорији алгоритама.

Перформансе израчунавања нису непосредно зависне од начина превођења, већ само од употребљене израчунљиве форме и евентуалних оптимизација. Израчунљивој форми израза је посвећена пажња у наредном одељку.

4.2.2. Извршно окружење

Програми се извршавају у оквиру извршног окружења интерпретатора. Најважније компоненте извршног окружења су:

- извршна форма програма;
- интерфејс за покретање програма;
- подсистем за израчунавање и
- интерфејс према ресурсима.

Извршна форма програма је тесно повезана како са начином извршавања програма тако и са управљањем меморијом и другим ресурсима. У случају *Wafl* интерпретатора обликована је нова форма израза која представља комбинацију редуцибилног графа и *G*-машине [John1984].

Сам интерфејс за покретање програма се разликује од контекста извршног окружења и може имати различите карактеристике. Једна врста интерфејса је имплементирана за извршавање програма на Веб серверу, а друга за извршавање програма из командне линије. Подсистем за израчунавање повезује специфичности модела израчунљиве форме израза и подсистеме за управљање меморијом и другим ресурсима. У оквиру подсистема за управљање радном меморијом налази се и аутоматски сакупљач отпадака. Интерфејс према ресурсима чине интерфејси према датотекама, бази података и другим ресурсима који чине окружење.

Програмски језик *Wafl* је имплементиран ради развоја Веб услуга, па је и подсистем за израчунавање прилагођен раду у одговарајућим условима. Слика 2 илуструје архитектуру извршног окружења за Веб услуге написане на програмском језику *Wafl*.

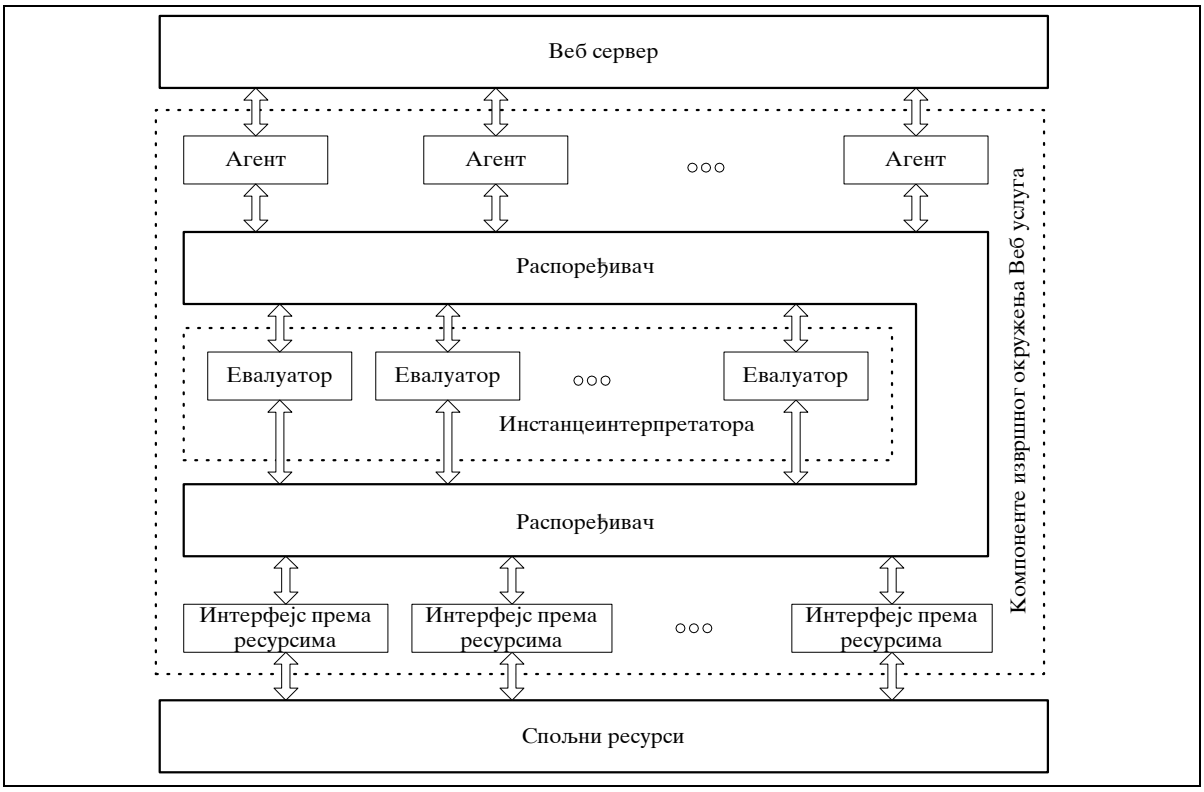
Веб сервер има улогу покретача програма. Он прима захтеве од корисника Веб услуге и покреће одговарајуће програме ради израчунавања резултата које је потребно проследити корисницима. Веб сервер покреће процесе који имају улогу *агената*. У једном тренутку може да постоји велики број агената. Један агент је задужен за тачно једно израчунавање. Агенти представљају једино место на коме Веб сервер и интерпретатор остварују међусобну комуникацију. Имплементацијом одговарајуће врсте агената се омогућава да Веб сервера употребљава програме на *Wafl*-у.

Агент има сасвим једноставна и уска задужења. Први његов посао је да проследи распоређивачу захтев за израчунавање програма. Тај захтев обухвата податке којима се идентификују Веб услуга и програм, као и одговарајуће параметре. Након упућивања захтева агент прелази у стање чекања на резултат. Из стања чекања ће се пробудити када му распоређивач пошаље резултат израчунавања. Након тога агент обавља свој други задатак – шаље резултат израчунавања Веб серверу.

Улога распоређивача је да управља радом система и остварује размену података између различитих компоненти система. Прва улога распоређивача је да захтеве пристигле од агената распоређује евалуаторима, као и да резултате израчунавања

прослеђује одговарајућим агентима. Распоређивач при томе прави нове и затвара старе и непотребне процесе евалуатора.

Друга улога распоређивача је да евалуаторима ставља на располагање одговарајуће интерфејсе према спољним ресурсима. У том контексту, спољни ресурси обухватају систем датотека, базе података, кориснички интерфејс и све друге видове ресурса који могу бити присутни у рачунарском систему.



Слика 2: Процеси који чине извршно окружење за Wафл програме у условима извршавања Веб услуга

Евалуатор је процес који израчунава програме. Број активних евалуатора зависи од конфигурације система и процене оптерећења и перформанси. Распоређивач по потреби формира инстанце евалуатора и прослеђује им захтеве пристигле од агената. Један евалуатор у једном тренутку може израчунавати највише један програм. Ако програм има захтеве у односу на спољне ресурсе, евалуатор од распоређивача тражи одговарајући интерфејс и користи услуге добијеног интерфејса.

Један програм се током свог израчунавања може израчунавати на више различитих евалуатора. У случају да евалуатор процени да ће израчунавање неког захтева, који је упућен путем интерфејса према ресурсима, трајати довољно дуго, он може да стави програм у стање чекања на ресурсе и да се пријави распоређивачу да је слободан за израчунавање других програма. Када интерфејс према ресурсима заврши посао и врати податке, распоређивач може наставак израчунавања одговарајућег програма да повери било ком слободном евалуатору.

4.3. Подсистем за распознавање типова

Подсистем за распознавање типова уграђен у експерименталну имплементацију програмског језика *Wafl* почива на Хиндли-Милнеровом алгоритму. Због неких специфичности програмског језика *Wafl* овај алгоритам је морао да претрпи одређено прилагођавање, али није било увођења суштинских новина. Основна разлика у односу на начине распознавања типова у неким другим језицима је у независном распознавању типа сваке појединачне употребе имена у изразу, чиме је омогућено да се исто име у једном изразу употребљава са различитим конкретним типовима. На тај начин се омогућава распознавање типова неких сложенијих израза (Пример 4).

```
// типови функције f у овом примеру су:
// - полиморфна дефиниција: '1 -> '1
// - 1. употреба у изразу:  ((Int->Int)->(Int->Int))->((Int->Int)->(Int->Int))
// - 2. употреба у изразу:  (Int->Int)->(Int->Int)
// - у функцији g:          Int->Int

f(f)(g)(2)
where {
    f(x) = x;
    g(x) = f(x+1);
};
```

Пример 4: Израз у коме се једно име употребљава са различитим типовима

Основне компоненте имплементираног алгоритма распознавања типова су *упаривање типова делова израза* и *унификација типова*. Упаривањем типова делова израза се анализирају изрази и постављају хипотезе о слагању типова одређених делова израза. Затим се на основу постављених хипотеза праве типовне једначине (енгл. *type equation*) које су њима одређене. Алгоритам унификације типова тражи најопштије решење добијеног система типовних једначина и истовремено проверава да ли је систем типовних једначина сагласан или противречан (тј. да ли има или нема решење). Оваква подела на фазе има одређене сличности са алгоритмом представљеним у [Heer2002].

Тип сваког дефинисаног имена се одређује помоћу анализе типа израза којим је име дефинисано и помоћу одговарајуће околине. Основни корак представља распознавање типа дефиниције у њеној сопственој околини, тј. без употребе имена која су дефинисана у ширим околинама. Поступак препознавања свих типова у програмском модулу се своди на рекурзивну примену основног корака, по дубини у односу на структуру израза, најпре од најдубље дефинисаних имена, па постепено, према читавом програмском модулу. Након сваког успешног препознавања се на основу свих препознатих имена проширује скуп аксиома који се користи у наредним препознавањима.

При распознавању типа дефиниције у сопственој околини:

- не употребљавају се имена која су дефинисана у спољашним околинама;
- у таквом распознавању учествују у виду аксиома све поддефиниције чији је тип распознат у њиховој сопственој околини;
- све поддефиниције чији тип није распознат у сопственој околини се распознају истовремено са основном дефиницијом;

- такво распознавање може да произведе позитиван резултат искључиво за дефиниције у којима се не употребљавају друга дефинисана имена, или се употребљавају само сопствене поддефиниције:
  - елементаран пример случаја који се не може овако распознати је узајамна рекурзија: ако се две функције међусобно употребљавају, тада ниједна од њих не може бити довољно добро анализирана у сопственој околини;
- само резултат распознавања типа дефиниције у сопственој околини (наравно, уколико је такво распознавање уопште могуће) обезбеђује да се увек сигурно добије најопштији могући тип дефиниције и
- захтеви за ресурсима су мањи него при распознавању у сложенијој околини.

Ако распознавање дефиниције не успе у њеној сопственој околини, онда ће се покушати распознавање њеног типа најпре у непосредној околини у којој се она налази, тј. истовремено са распознавањем типа основне дефиниције чију поддефиницију она представља. Ако ни то не успе, наставља се даље, рекурзивно, све док се не дође на ниво програмског модула. Ако ни то не успе, извештава се о одговарајућим типовним грешкама.

Упаривање типова делова израза

Алгоритам упаривања типова делова израза је компонента алгоритма распознавања типова која је тесно повезана са алгоритмом повезивања имена у изразима. При сваком покушају повезивања имена које има више потенцијалних типова, о исправности употребе имена се одлучује на основу сагласности типова. Због тога су поступци повезивања имена и упаривања типова делова израза блиско везани и могу се посматрати као једна сложена фаза у превођењу програма.

Упаривање типова делова израза зависи од врсте израза који је у питању, при чему се разликују три основне категорије израза:

- дефиниције функција;
- позиви функција и други сложени изрази (укључујући и операторске изразе и друге сложене изразе који имају семантику примене функције, као на пример израз `if`) и
- литералне константе и непосредно наведена имена.

Распознавање типа резултата функције практично се изводи распознавањем типа резултата израза који дефинише функцију. Распознавање типа аргумената се одвија постепено током распознавања типа резултата израза, тако што се на почетку претпостави да аргументи имају различите непознате полиморфне типове, који се затим постепено препознају и редукују при свакој појави неког од аргумената у изразу.

Ознаке

У даљем тексту се употребљавају ознаке:

- Tip(*izraz*) – пресликавање датог израза програмског језика у одговарајући тип;
- TipRes(*fun*) –пресликавање функцијског израза у тип резултата функције и

- TipArg(*fun*,*n*) –пресликавање функцијског израза у тип *n*-тог аргумента те функције.

Типовне променљиве се означавају у облику *'n*, где је *n* неки природан број. Број *n* се назива индексом типовне променљиве. Начелно, типовна променљива има општи домен.

Полиморфни типови

Полиморфни типови су типови који представљају унију два или више типова. У оквиру њиховог записа се наводе типовне променљиве.

Ако полиморфан тип представља унију свих типова, онда је у питању прост полиморфан тип. Запис простог полиморфног типа се састоји само од одговарајуће типовне променљиве.

Тип није полиморфан ако представља неки егзактан тип, ма колико да је сложен. Неполиморфни типови у запису немају типовне променљиве.

Запис полиморфног типа је канонизован ако се у њему појављују све типовне променљиве од *'1* до *'n*, за неко *n*, и при томе за сваки пар природних бројева *i* и *j*, таквих да важи $1 \leq i < j \leq n$, важи да се прво појављивање типовне променљиве *'i* налази пре првог појављивања типовне променљиве *'j* у запису типа. Два полиморфна типа су једнака ако и само ако су им канонизовани записи једнаки.

У неким случајевима у алгоритму је потребно правити независне копије полиморфних типова, како би се могли употребљавати и проверавати независно од других инстанци. На пример, полиморфан оператор > може у истом изразу да буде примењен најпре на целе бројеве а затим на ниске. Независне копије полиморфних типова се називају инстанце.

Нова инстанца полиморфног типа се прави тако што се у копији канонизованог записа полиморфног типа индекси свих променљивих увећавају за вредност највећег индекса полиморфног типа који је то тог тренутка употребљен у том основном кораку проверавања типова.

Аксиоме

Аксиоме су типовна тврђења за која се априори претпоставља да важе. Имају облик:

$$\text{Tip}(ime) = T$$

где је *T* тип који одговара имену *ime*.

Почетни скуп аксиома се прави на основу скупа уграђених дефинисаних имена програмског језика. На пример:

$$\text{Tip}(if) = \text{Bool} * '1 * '1 \rightarrow '1$$

Током поступка проверавања типова, за свако име за које се поуздано установи тип додаје се нова аксиома која се затим употребљава при наредним корацима распознавања типова.

Неполиморфне аксиоме се непосредно употребљавају у поступку установљавања типова, док се полиморфне аксиоме користе посредно, у облику нових инстанци.

Правило дефиниције

Нека је у поступку употребљено *n* типовних променљивих.

На основу дефиниције облика:

$$ime = izraz$$

постављају се хипотезе облика:

$$\text{Tip}(ime) = '(n+1)$$
$$\text{Tip}(izraz) = '(n+1)$$

где је *ime* име које се дефинише датим изразом *izraz*, а $'(n+1)$ је нова типовна променљива.

Ако дефиниција представља дефиницију функције облика:

$$ime(arg1,...,argk) = izraz$$

Онда се постављају хипотезе облика:

$$\text{Tip}(ime) = '(n+1) * ... * '(n+k) \text{ -> } '(n+k+1)$$
$$\text{Tip}(izraz) = \text{Tip}(izraz) = '(n+k+1)$$

Правило сложеног израза

Изрази имају структуру терма. Без обзира на то да ли се израз записује у облику примене функције или примене инфиксног оператора, сваки неелементаран израз се може представити у облику:

$$izraz = fun(arg1,...,argk)$$

За сваки сложен израз се постављају хипотезе:

$$\text{Tip}(izraz) = \text{TipRes}(fun)$$
$$\text{Tip}(argk) = \text{TipArg}(fun,1)$$
$$\dots$$
$$\text{Tip}(arg1) = \text{TipArg}(fun,k)$$

Затим се рекурзивно примењују правила на све изразе који одговарају аргументима.

Правило простог израза

Елементарни изрази представљају или литералне константе или имена:

$$izraz = konstanta$$
$$izraz = ime$$

За свако појављивање константе се поставља по хипотеза облика:

$$\text{Tip}(izraz) = T_{konstante}$$

где је $T_{konstante}$ тип одговарајуће константе.

У случају појављивања имена:

- ако је тип имена утврђен неком аксиомом као неполиморфан, онда се не уводи нова хипотеза;

- ако је тип имена утврђен неком аксиомом као полиморфан, онда се уводи хипотеза да је тип израза (тј. имена) нова инстанца одговарајућег полиморфног типа:

$$\text{Tip}(izraz) = T_{inst}$$

- ако име представља *k*-ти аргумент функције чији се тип проверава, онда се уводи хипотеза да тип израза одговара очекиваном типу аргумента:

$$\text{Tip}(izraz) = T_{arg}$$

- ако тип имена није утврђен аксиомом, онда је управо у току провера и тог типа, па се уводи хипотеза по којој је тип израза једнак претпостављеном типу имена:

$$\text{Tip}(izraz) = T_{ime}$$

Алгоритам

Алгоритам упаривања типова подразумева да се уз сваки израз води и контекст у коме се појављује, тако да се два синтаксно идентична подизраза, која се појављују на два места у програму, не сматрају за идентичне изразе. То је неопходно да би полиморфни изрази могли у различитим контекстима да имају различите типове.

1. На основу дефиниције чији је тип потребно препознати (или скупа дефиниција чији се типови препознају) и правила дефиниције се направи почетни скуп хипотеза;
2. Понавља се:
   2.1 Тражи се хипотеза настала применом правила дефиниције или правила сложеног израза на коју није примењено ни правило сложеног израза ни правило простог израза;
   2.2 Ако постоји:
      2.2.1 На њу се примењује правило сложеног израза или правило простог израза;
      2.2.2 Наставља се од корака 2.1;
   2.3 Иначе, ако не постоји, прекида се понављање и наставља се од корака 3;
3. Од сваког пара хипотеза са једнаком левом страном прави се по једна типовна једначина.
4. Резултат представља скуп типовних једначина.

Алгоритам 1: Основни корак алгоритма упаривања типова.

Из описа алгоритма и наведених правила, очигледно следи да се за сваку примену неког подизраза као аргумента неке функције или операције добија по једна типовна једначина. Што је сложенији израз којим се дефинише неко име, број једначина које ће се добити је већи.

Пример примене алгоритма упаривања типова, као и читавог алгоритма распознавања типова, је приложен у додатку *Пример примене алгоритма распознавања типова*, на страни 201.

Решавање система типовних једначина

Систем типовних једначина се решава алгоритмом унификације типова. Алгоритам унификације типова подразумева два улазна типа, чијом се унификацијом покушава добити њихов најшири пресек, који на крају представља резултат алгоритма. Уколико не

постоји пресек датих типова, алгоритам препознаје да типови нису међусобно сагласни. Уколико пресек постоји, алгоритам унификације гарантује да ће се као резултат добити најопштије могуће решење.

Унификација типова представља итеративно примењивање корака у којима се истовремено установљава да ли су дати типови међусобно сагласни и постепено распознају прости полиморфни типови. У једном кораку распознавања неки прост полиморфан тип може бити:

- замењен неполиморфним типом;
- замењен полиморфним типом који није прост или
- изједначен са неким другим простим полиморфним типом.

Сваким изведеним кораком распознавања се сужава скуп могућих типова који одговарају датом скупу имена или израза. Због тога се алгоритам унификације типова назива и алгоритмом *редукције полиморфних типова*.

Таблица вредности типовних променљивих

У алгоритму за решавање система типовних једначина се употребљава таблица вредности типовних променљивих. Таблица вредности типовних променљивих садржи по елемент за сваку типовну променљиву која се појављује у датом скупу типовних једначина.

1. Таблица вредности типовних променљивих има по елемент за сваки полиморфан тип (тј. за сваку типовну променљиву) који се појављује у типовним једначинама;
2. Почетна вредност сваке типовне променљиве у таблици је *недефинисано*.

Алгоритам 2: Иницијализација садржаја таблице вредности типовних променљивих.

Читање вредности из таблице се одвија рекурзивно, све док се не добије тип који се не може даље упростити. При томе се води рачуна да се не уђе у бесконачну рекурзију.

1. Као почетни тип T се претпоставља вредност типовне променљиве $'n$ из таблице вредности типовних променљивих;
2. Ако је тип T дефинисан и у оквиру типа T постоји типовна променљива за коју је дефинисано пресликавање, а која је различита од типовне променљиве $'n$ чија се вредност чита:
   2.1 Свако појављивање такве типовне променљиве у типу T се замењује вредношћу која јој одговара у таблици, а која се установљава рекурзивном применом овог алгоритма на таблицу која не обухвата елемент који се односи на типовну променљиву $'n$;
   2.2 Део типа који је управо замењен се искључује из даљег замењивања;
   2.3 Понавља се корак 2;
3. Иначе, поступак је довршен и тип T представља резултат читања вредности типовне променљиве из таблице.

Алгоритам 3: Читање вредности типовне променљиве из таблице вредности типовних променљивих.

Употреба таблице се може заменити одговарајућим еквивалентним поступком замењивања свих појављивања неке типовне променљиве у хипотезама и једначинама датом вредношћу те типовне променљиве.

Алгоритам унификације типова

Унификација типова проверава сагласност датих двају типова сводећи их на највећи заједнички пресек. При томе се, у случају да је бар један од датих типова полиморфан, потенцијално сужава опсег вредности типовних променљивих које се појављују у оквиру полиморфних типова. Унификација типова почива на скупу правила за унификацију различитих категорија типова.

Сви типови података се интерно, на нивоу подсистема за распознавање типова, могу поделити на:

- просте полиморфне типове: *'n, 'm, 'k, ...* ;
- просте типове: Bool, Integer, Float и String;
- комбиноване типове: Numeric[*t*], Value[*t*], Prime[*t*].
- сложене типове:
  - торке: Tuple[*T1*, ..., *Tn*], Tuple[*T1*, ..., *Tn*][*x*];
  - слогове: Record[attr1:*T1*, ..., attr*n*:*Tn*], Record[attr1:*T1*, ..., attr*n*:*Tn*][*x*];
  - колекције: List[*t*], Map[*t1*][*t2*];
- и функцијске типове.

Унификација је операција U која за два типа *T1* и *T2* и скуп вредности типовних променљивих *Env* (који се представља одговарајућим садржајем таблице вредности типовних променљивих) установљава највећи могући пресек *T* и мења скуп вредности типовних променљивих сводећи их на конкретније типове у најмањој могућој мери у којој је то неопходно да би пресек *T* постојао. Употребљава се нотација:

$$U(T1, T2; Env) = T; Env^*$$

За проширивање или ажурирање околине *Env* новом вредношћу *T* типовне променљиве *'n* употребљава се ознака *Env*[*'n=T*].

Унификација је у суштини комутативна у односу на типове који се унификују, мада у резултујућем типу и новој проширеној околини новоуведене типовне променљиве могу бити различито именоване.

При унификацији типова се подразумева да су типови који се унификују већ сведени у односу на садржај таблице типовних променљивих, тј. све могуће замене типовних променљивих помоћу таблице су већ обављене.

Основна правила за унификацију типова су:

- два једнака типа се увек успешно унификују:

$$U(T, T; Env) = T; Env$$

- прост полиморфан тип *T1='n* се успешно унификује са простим полиморфним типом *T2='m*, при чему је резултат унификације прост полиморфан тип са мањим индексом:
  - вредност типовне променљиве са већим индексом у таблици се поставља на прост полиморфан тип са мањим индексом;

$$\frac{i > 0}{\mathrm{U}('n, '(n+i); Env) = 'n; Env['(n+i)='n]}$$

- прост полиморфан тип *'n* се успешно унификује са сваким типом *T* који није просто полиморфан, при чему је резултат унификације управо тип *T*;
  - вредност типовне променљиве *'n* у таблици се поставља на *T*

$$\frac{T \text{ није прост полиморфан тип}}{\mathrm{U}('n, T; Env) = T; Env['n=T]}$$

- два комбинована типа се унификују ако и само ако домени оба комбинована типа имају пресек, при чему је резултат комбиновани тип чији је домен пресек полазних домена а параметар резултат унификације параметара;

$$\frac{\begin{array}{rcl} C & = & C1 \cap C2 \\ \mathrm{U}('n, 'm; Env) & = & 'm; Env^* \end{array}}{\mathrm{U}(C1['n], C2['m]; Env) = C['k]; Env^*}$$

  - уколико се као резултат појави сингуларан домен, примењује се још један корак унификације типовне променљиве параметра са одговарајућим простим типом *T*:

$$\frac{C = \{ T \}}{C['n]; Env = \mathrm{U}('n, T; Env)}$$

- комбинован тип *C['n]* и прост тип *T* се унификују ако и само ако прост тип припада домену комбинованог типа, при чему је резултат управо дати прост тип:

$$\frac{T \in C}{\mathrm{U}(T, C['n]; Env) = \mathrm{U}('n, T; Env)}$$

- сложени тип *S[T1]* може да се унификује са одговарајућим сложеним типом *S[T2]* тако што се унификују њихови типови елемената:

$$\frac{\mathrm{U}(T1, T2; Env) = T; Env^*}{\mathrm{U}(\mathrm{S}[T1], \mathrm{S}[T2]; Env) = \mathrm{S}[T]; Env^*}$$

- два функцијска типа *T1=(T1.1 *... * T1.*n -> *T1.0)* и *T2=(T2.1 *... * T2.*n -> *T2.0)* се унификују ако и само ако су исте арности и успешно се унификују типови њихових резултата и типови парова одговарајућих аргумената:

$$\frac{\begin{array}{c} \mathrm{U}(T1.0, T2.0; Env) = T3.0; Env_0 \\ \mathrm{U}(T1.1, T2.1; Env_0) = T3.1; Env_1 \\ \dots \\ \mathrm{U}(T1.\mathrm{n}, T2.\mathrm{n}; Env_{\mathrm{n}-1}) = T3.\mathrm{n}; Env_{\mathrm{n}} \end{array}}{\mathrm{U}(T1, T2; Env) = (T3.1 \ast \dots \ast T3.\mathrm{n} \rightarrow T3.0); Env_{\mathrm{n}}}$$

- два типа торки *T1*=Tuple[*T1.1*, ..., *T1*.n] и *T2*=Tuple[*T2.1*, ..., *T2*.n] се унификују ако и само ако су исте арности и успешно се унификују типови њихових елемената:

$$\frac{\begin{array}{r}U(T1.1, T2.1; Env) = T3.0; Env_1 \\ \dots \\ U(T1.n, T2.n; Env_{n-1}) = T3.n; Env_n\end{array}}{U(T1, T2; Env) = \text{Tuple}[T3.1, ..., T3.n]; Env_n}$$

- непотпуни тип торки *T1*=Tuple[*T1.1*, ..., *T1*.n]['*i*] се унификује са типом торки *T2*=Tuple[*T2.1*, ..., *T2*.m] који је бар исте дужине ако и само ако се успешно унификују редом типови њихових елемената и одговарајуће проширење:

$$\frac{\begin{array}{r}n \le m \\ U(T1.1, T2.1; Env) = T3.1; Env_1 \\ \dots \\ U(T1.n, T2.n; Env_{n-1}) = T3.n; Env_n \\ U('i, \text{Tuple}[T2.(n+1), ..., T2.m]; Env_n) = \text{Tuple}[T3.(n+1), ..., T3.m]; Env^*\end{array}}{U(T1, T2; Env) = \text{Tuple}[T3.1, ..., T3.n, T3.(n+1), ..., T3.m]; Env^*}$$

- непотпуни тип торки *T1*=Tuple[*T1.1*, ..., *T1*.n]['*i*] се унификује са непотпуним типом торки *T2*=Tuple[*T2.1*, ..., *T2*.m]['*j*] који је бар исте дужине ако и само ако се успешно унификују редом типови њихових елемената и њихова проширења:

$$\frac{\begin{array}{r}n \le m \\ U(T1.1, T2.1; Env) = T3.1; Env_1 \\ \dots \\ U(T1.n, T2.n; Env_{n-1}) = T3.n; Env_n \\ U('i, \text{Tuple}[T2.(n+1), ..., T2.m]['j]; Env_n) = \text{Tuple}[T3.(n+1), ..., T3.m]['j]; Env^*\end{array}}{U(T1, T2; Env) = \text{Tuple}[T3.1, ..., T3.n, T3.(n+1), ..., T3.m]['j]; Env^*}$$

- два слоговна типа *T1*=Record[attr*1.1*:*T1.1*, ..., attr*1.n*:*T1*.n] и *T2*=Record[attr*1.1*:*T2.1*, ..., attr*2.n*:*T2*.n] се унификују ако и само ако су исте арности, сви атрибути су им редом (у лексикографском поретку) са међусобно истим именима и успешно се унификују њихови типови атрибута:

$$\frac{\begin{array}{r}\text{attr}1.1 = \text{attr}2.1 \\ U(T1.1, T2.1; Env) = T3.1; Env_1 \\ \dots \\ \text{attr}1.n = \text{attr}2.n \\ U(T1.n, T2.n; Env_{n-1}) = T3.n; Env_n\end{array}}{U(T1, T2; Env) = \text{Record}[\text{attr}1.1:T3.1, ..., \text{attr}1.n:T3.n]; Env_n}$$

- непотпун слоговни тип *T1*=Record[attr1.1:*T1.1*, ..., attr1.n:*T1*.n]['*i*] се унификује са слоговним типом *T2*=Record[attr2.1:*T2.1*, ..., attr2.m:*T2*.m] ако и само се успешно унификују типови њихових одговарајућих елемената и одговарајуће проширење:

$$\frac{\begin{array}{r}n \le m \\ \text{attr}1.1 = \text{attr}2.1 \\ U(T1.1, T2.1; Env) = T3.1; Env_1 \\ \dots \\ \text{attr}1.n = \text{attr}2.n \\ U(T1.n, T2.n; Env_{n-1}) = T3.n; Env_n\end{array}}{}$$

$$\frac{U('i, \text{Record}[\text{attr}2.(n+1):T2.(n+1), ..., \text{attr}2.m:T2.m]; Env_n) = \text{Record}[\text{attr}2.(n+1):T3.(n+1), ..., \text{attr}2.m:T3.m]; Env^*}{U(T1, T2; Env) = \text{Record}[\text{attr}2.1:T3.1, ..., \text{attr}2.1:T3.n, \text{attr}2.1:T3.(n+1), ..., \text{attr}2.1:T3.m]; Env^*}$$

- два непотпуна слоговна типа $T1$=Record[attr1.1:$T1.1$, ..., attr1.n:$T1$.n]['i] и $T2$=Record[attr2.1:$T2.1$, ..., attr2.m:$T2$.m] се успешно унификују ако и само ако им се успешно унификују типови свих заједничких атрибута, и одговарајући типови проширења са специфичним атрибутима:

$$\frac{\begin{array}{l}U(\text{Record}[\text{attr}1.1:T1.1, ..., \text{attr}1.k:T1.k], \text{Record}[\text{attr}2.1:T2.1, ..., \text{attr}2.k:T2.k]; Env) = \text{Record}[\text{attr}1.1:T3.1, ..., \text{attr}1.k:T3.k]; Env^* \\ \{\text{attr1.(k+1)}, ..., \text{attr1.n}\} \cap \{\text{attr2.(k+1)}, ..., \text{attr2.m}\} = \varnothing\end{array}}{\begin{array}{l}U(T1, T2; Env) = \text{Record}[\text{attr}1.1:T3.1, ..., \text{attr}1.k:T3.k, \\ \quad \text{attr}1.(k+1):T1.(k+1), ..., \text{attr}1.n:T1.n, \\ \quad \text{attr}1.(k+1):T1.(k+1), ..., \text{attr}2.m:T2.m]; \\ \quad Env['i = \text{Record}[\text{attr}2.(k+1), ..., \text{attr}2.m]['l]] \\ \quad ['j = \text{Record}[\text{attr}1.(k+1), ..., \text{attr}2.n]\end{array}}$$

- унификација било која друга два типа $T1$ и $T2$ не успева.

Примена правила се, према потреби, изводи рекурзивно.

Алгоритам решавања система типовних једначина

Као што је већ наглашено, алгоритам решавања система типовних једначина почива на употреби таблице вредности типовних променљивих и алгоритму унификације типова:

1. Направи се таблица пресликавања вредности типовних променљивих Env:
   1.1 Низ има по један елемент за сваки полиморфан тип (тј. за сваку типовну променљиву) који се појављује у типовним једначинама;
   1.2 Почетна вредност сваког пресликавања је *недефинисано*;
2. Примени се унификација:
   2.1 За сваку типовну једначину, редом:
      2.1.1 Свака типовна променљива у једначини се замени вредношћу те типовне променљиве која се чита из таблице вредности типовних променљивих;
      2.1.2 Једначина је облика $T1 = T2$. Примењује се унификација типова $T1$ и $T2$ према наведеним правилима и уз употребу таблице пресликавања типовних променљивих Env;
      2.1.3 Ако је унификација успела и $U(T1,T2;Env)=T;Env^*$, онда се ажурира таблица пресликавања вредности типовних променљивих $Env \leftarrow Env^*$;
      2.1.4 Иначе, ако унификација није успела, поступак се прекида и извештава се да типови нису сагласни;
3. За сваки претпостављени тип имена дефиниције (из хипотеза насталих правилима дефиниција):
   3.1 Свака типовна променљива у типу се замени вредношћу те типовне променљиве која се чита из таблице вредности типовних променљивих (Алгоритам 3);
4. Тако замењене вредности претпостављених типова представљају препознате типове дефиниција.

Алгоритам 4: Решавање система типовних једначина помоћу таблице вредности типовних променљивих и алгоритма унификације типова.

Ограничења подсистема за рад са типовима

Основно ограничење подсистема за рад са типовима у програмском језику *Wafl* је везано за просторе имена. Да би се успешно препознао тип функција које су узајамно рекурзивне, оне морају бити у истом простору имена. Чињеница да највећи простор имена представља датотека имплицира да не сме постојати узајамна рекурзија међу функцијама из разних датотека.

Иако ово ограничење није посебно значајно, чак се и оно може превазићи. Један начин превазилажења је да се једна од узајамно рекурзивних функција (или више њих) предаје другој као аргумент.

5. Постојећа повезивања програмских језика и база података

5.1. Библиотеке функција

При повезивању процедуралних програмских језика и база података уобичајено је да се употребљавају различите библиотеке функција. Данас је у редовној употреби више таквих библиотека. Ако се разматрање ограничи на библиотеке које омогућавају униформан приступ подацима похрањеним у различитим системима за управљање базама података, број значајних библиотека се значајно смањује.

Једна од најважнијих библиотека функција је *SQL CLI* (енгл. *SQL Call Level Interface*). *SQL CLI* је саставни део стандарда *SQL*-а. Представља програмски интерфејс за писање клијентских програма који остварују комуникацију са СУБП-ом. Подржава и статички и динамички рад.

Иако стандардизована, ова библиотека не гарантује потпуну преносивост програма. Она се имплементира независно за сваки конкретан СУБП, па је због тога неопходно понављати превођење програма за различите СУБП-ове. Таква архитектура је допустила и постојање одређених разлика у имплементацијама библиотека, због чега прелазак са једног на други СУБП није у потпуности безболан.

Једна од најзаступљенијих актуелних библотека функција је *ODBC* [Geig1995]. *ODBC* је програмски интерфејс за тзв. отворено повезивање са базама података (енгл. акроним за *Open DataBase Connectivity*). *ODBC* је обликован на основу више различитих имплементација библиотека *SQL CLI*. Изворно је у написан за програмски језик *C*, али постоје имплементације и за друге програмске језике. Основна мана *ODBC*-а је што је ограничен на динамички рад са подацима. Поред тога, практично још од 1995. године није усклађен са новијим верзијама стандарда *SQL*-а.

Архитектура *ODBC*-а прописује да се имплементира јединствена библиотека, која се драјверима ослања на конкретне базе података. Практично сви савремени комерцијални системи за управљање базама података имају одговарајуће драјвере. Такође, за практично сваки савремен оперативни систем постоје имплементације библиотеке

ODBC. Због тога је ниво преносивости програма који користе ову библиотеку веома висок.

ODBC је послужио као основа за развој више различитих интерфејса који су специфични за различита развојна окружења, као што су, на пример, *OLEDB* и *ADO*. *OLEDB* и *ADO* су објектно оријентисани интерфејси вишег нивоа за фамилију оперативних система *Windows* чија се имплементација ослања на *ODBC*.

По узору на *SQL CLI* и *ODBC* је дефинисан интерфејс *JDBC* [Hami1996] за приступање базама података из програмског језика *Java*, који углавном дели њихове особине.

5.2. Угњеждени *SQL*

Угњеждени *SQL* (енгл. *Embedded SQL*) је саставни део стандарда упитног језика *SQL*. Угњеждени *SQL* се одликује скупом основних правила о начину употребе елемената упитног језика *SQL* у програмима писаним на различитим програмским језицима, при чему се неки елементи синтаксе прилагођавају конкретним програмским језицима.

Угњеждени *SQL* представља пример концептуалног приступа проблему интеграције програмског и упитног језика. Делови програма написани на упитном језику могу се релативно једноставно, а често и без икаквих измена, употребљавати у програмима написаним на различитим програмским језицима. У подржане програмске језике спадају *C*, *C++*, *Cobol*, *PL/I*, *Smalltalk*, *Java*, *Fortran* и други. Конкретне имплементације РСУБП-ова подржавају различите скупове програмских језика.

Угњеждени *SQL* омогућава непосредну употребу елемената упитног језика у оквиру програма писаних на програмском језику. Конструкције упитног језика се могу употребљавати како за читање тако и за мењање садржаја базе података. Постоји строга провера типова података који се размењују између програма и базе података[19].

Примена угњежденог *SQL*-а почива на следећим концептима и правилима:

- Програмски језик на коме се пише програм који се повезује са базом података помоћу угњежденог *SQL*-а назива се *матични језик*.
- Угњеждени *SQL* се имплементира помоћу претпроцесора за одговарајући програмски језик.
  - Све конструкције угњежденог *SQL*-а представљају претпроцесорске директиве, које се претпроцесирањем преводе у одговарајуће конструкције матичног језика.
  - Све конструкције угњежденог *SQL*-а почињу кључним речима `EXEC SQL`[20], иза чега следи наредба *SQL*-а записана на уобичајен начин.
- Размена података између програма и базе података се одвија посредством *матичних променљивих*. Матичне променљиве су посебне променљиве програмског језика које се могу употребљавати у оквиру упита на *SQL*-у.

[19] Наравно, у мери у којој то допуштају синтакса и семантика самог програмског језика.

[20] У неким конкретним имплементацијама се уместо префиксног пара кључних речи `EXEC SQL` употребљавају неке друге конструкције. На пример, у случају система *DB2* и *IBM*-овог скрипт језика *REXX*, угњеждене наредбе се записују у облику: `CALL SQLEXEC 'sql statement'`.

  - Матичне променљиве се декларишу у посебно означеним одељцима, који почињу директивом `EXEC SQL BEGIN DECLARE SECTION`, а завршавају се директивом `EXEC SQL END DECLARE SECTION`.
  - Подржани су само прости типови података. У пракси, допуштени типови матичних променљивих зависе од конкретне имплементације РСУБП-а и конкретног програмског језика.
  - У конструкцијама матичног програмског језика матичне променљиве се употребљавају на исти начин као и све друге променљиве програмског језика.
  - У конструкцијама угњежденог *SQL*-а матичне променљиве се употребљавају са префиксом „:“ (симбол „две тачке“).
- Трансакцијама се управља помоћу одговарајућих наредби *SQL*-а, а у складу са правилима синтаксе угнежденых наредби.

Ограничење угњежденог *SQL*-а је у једносмерности везе коју пружа. Да би могла да се по својим основним могућностима пореди са решењима која пружају пуну интеграцију програмског и упитног језика, концептуална интеграција мора бити општија, пре свега тако да омогући употребу програма или делова програма писаних на програмском језику у оквиру упита писаних на упитном језику. Елементи које угњеждени *SQL* не обухвата, а који би допринели широј (али не обавезно и тешњој) интеграцији, су пре свега:

- похрањивање (и читање) сложених података програмског језика у (из) бази података и
- употреба елемената програмског језика у упитима на упитном језику.

5.3. Процедурални *SQL*

Од стандарда *SQL*:1999, *SQL* је допуњен процедуралним елементима. Том приликом је *SQL* и формално добио процедуре и функције које раде на страни СУБП-а, као и активне објекте (окидаче). У контексту интеграције програмских и упитних језика, најзначајнија је чињеница да је језик допуњен контролним структурама и променљивим, чиме је омогућено да се на *SQL*-у пишу програми за обраду података. На тај начин је *SQL* постао рачунски комплетан језик. Штавише, иста верзија стандарда је прописала и објектно оријентисана проширења језика *SQL*.

Стандардизација процедуралних елемената упитног језика је дошла са значајним закашњењем. У то време су практично све значајне имплементације РСУБП-а већ имале специфичне имплементације процедуралних елемената језика. Због тога и данас постоји велика неусаглашеност међу различитим имплементацијама. Последица је релативно низак ниво преносивости процедуралног кода.

Основни квалитети процедуралног *SQL*-а се огледају у могућности да се делови кода који се извршавају на страни СУБП-а развијају на упитном језику, који је по својој природи далеко ближи подацима него алтернативни програмски језици. Процедурални елементи су уграђени уз пуно поштовање трансакционих механизама упитног језика. На тај начин је омогућено да се трансакције над базом података имплементирају у

потпуности у оквиру СУБП-а. Поред трансакција, омогућено је и развијање процедура и функција које израчунавају резултате сложених упита са итеративном семантиком[21].

```
declare n integer;
set n = 10;
declare i,f integer with default 1;
while i<n do
    set i = i+1;
    set f = f*i;
end;
```

Пример 5: Пример итеративног кода на процедуралном *SQL*-у који израчунава факторијал целог броја *n*

Са друге стране, највећа слабост овако проширеног језика *SQL* је у самом концепту проширивања упитног језика до програмског језика, чиме је измењена природа језика и још више нарушена и иначе недоследна примена релационог модела. Резултат представља релативно нехомогену и неортогоналну комбинацију релационог и објектно-оријентисаног императивног језика. Колико год да је *SQL* квалитетан као упитни језик, у улози програмског језика је прилично незграпан и непрактичан. Један од основних проблема је у слабим могућностима повезивања објектно-релационог модела, на коме таква проширења почивају, са конкретним програмским језицима на којима се развијају апликације. Последице се осећају при покушајима да се на процедуралном *SQL*-у имплементирају неки сложенији алгоритми или програми.

Може се закључити да је процедурални *SQL* успешно проширио могућности језика за решавање проблема на страни СУБП-а, али да се није прибилижио решавању проблема на страни клијента. У том контексту, иако представља вид интегрисаног решења, процедурални *SQL* се мора и даље повезивати са програмским језицима при имплементирању информационих система.

5.4. Анализа Аткинсона и Бјунемана

Аткинсон и Бјунеман су анализирали проблеме типова и перзистентности у језицима база података [Atki1987]. Као резултат анализе обликовали су 9 принципа који сугеришу да би будућа истраживања и решења требало да обухватају:

1. Униформан језик, који не би постављао препреке за развој комплексних програма на језику базе података;
2. Обезбеђен механизам за контролу перзистентности који је независан од типова података;
3. Уграђене апстрактне типове података за представљање великих колекција података;
4. Механизме за полиморфно и ефикасно индексирање и читање података;

[21] Колико год да су релациони модел података и упитни језици примеренији за израчунавање различитих врста упита, постоје неки проблеми који по својој природи имају процесну семантику, па их је далеко једноставније решавати применом итеративних метода. Такви проблеми су, на пример, они код којих је при одређивању наредног реда резултата упита потребно узимати у обзир претходне редове резултата.

5. Програми или процедуре би требало да буду типизирани објекти чија се перзистентност третира на исти начин као у случају било којих других података;
6. Систем типова мора да омогући неки вид хијерархија;
7. Типови или модули морају да омогуће неки вид одложеног везивања (енгл. *dynamic binding*);
8. Статичку проверу типова у мери у којој је то могуће ускладити са (7) и
9. „Променљиве“ би требало да при дефинисању добијају вредност, која се касније не може мењати.

Иако њихови закључци то не наводе експлицитно, у принципима (7) и (8) је присутно имплицитно повезивање проблема са концептима строго типизираних функционалних програмских језика.

5.5. Функцијски интерфејс ка релационим базама података

Пример функцијског интерфејса ка релационим базама података, који се заснива на употреби угњежденог *SQL*-а је представљен у [Митић1995]. Ради се о интерфејсу који је дефинисан за програмски језик *Lisp*, али се већи део интерфејса може лако прилагодити другим функционалним програмским језицима. Ипак, подршка за рад са апстрактним типовима података у бази података је зависна од специфичне синтаксе програмског језика *Lisp*, па се не може непосредно пренети на други програмски језик.

Основни елемент интерфејса је унарна функција `EXECSQL`, која израчунава дати упит. У упиту се могу појавити матичне променљиве, па и матични изрази који се израчунавају пре израчунавања упита. Уколико је као упит наведена наредба *SQL*-а која мења податке, тада примена функције `EXECSQL` представља једну трансакцију. Уколико се у матичним изразима у некој трансакцији непосредно појављују друге апликације функције `EXECSQL`, тада оне не представљају нове трансакције већ су обухваћене широм трансакцијом у којој се употребљавају. Наредбе *SQL*-а за контролу трансакција нису непосредно подржане јер се управљање трансакцијама одвија имплицитно. Свака успешно окончана трансакција се потврђује, док се неуспешно окончане трансакције поништавају.

Представљени интерфејс се одликује великом изражајном моћи у смислу постављања упита. Поред произвољно сложених упита, могуће је једноставно дефинисати линеарне трансакције. Велики квалитет је подршка за рад са апстрактним типовима података. Штавише, у бази података се могу чувати и функције, па и упити на *SQL*-у.

Нешто је сложеније писање нелинеарних и посебно итеративних трансакција. Међутим, сам проблем дефинисања таквих трансакција је веома удаљен од основних концепата функционалног програмирања. У природи таквих проблема је експлицитна промена стања (тј. базе података), па је очекивана последица да функционална имплементација уводи неке отежавајуће околности у односу на императивну имплементацију.

5.6. Пројекат *Shadows*

Специфичан облик интеграције је направљен у оквиру пројекта *Shadows* на универзитету у Хелсинкију [Oksa2001]. Специфичност се огледа у ограничавању разматрања проблема на базе података у радној меморији рачунара. Резултат пројекта су нов алгоритам за управљање меморијом и уклањање отпадака, назван *Shades*, и перзистентан програмски језик *Shines*. Основна идеја при раду са меморијом је да се садржај никада не ажурира на истом месту, већ се увек обавља преписивање, по концепту *копирање-при-писању* (енгл. *copy-on-write*).

Такав приступ управљању меморијом је у складу са основним концептима функционалног програмирања. Програмски језик *Shines* је у основи функционалан програмски језик са елементима перзистентних операција. Нису представљене конструкције језика које се односе на управљање трансакцијама. Практична примена би захтевала неке даље дораде језика због тога што су неке уобичајене могућности релативно закомпликоване – на пример, није могућа елементарна рекурзија, већ мора да се изражава уз експлицитно преношење рекурзивне функције. Рад са функцијама вишег реда је могућ али се и он остварује посредно.

5.7. Систем *Kleisli*

Један од значајних покушаја успостављања пуне интеграције упитног и програмског језика представља систем *Kleisli* [Wong2000]. Систем *Kleisli* почива на програмском језику *SML* [Gilm1997], на коме је и написан (имплементација *SML/NJ* [Puce2001]). Систем обухвата упитни језик *CPL* (енгл. *Collection Programming Language*). Упитни језик је обликован као интегрисано проширење програмског језика *SML*. Резултати упита су подаци који се непосредно могу даље обрађивати у програму. Семантика *CPL*-а почива на угњежденом релационом рачуну (енгл. *Nested Relational Calculus – NRC* [Bune1995]).

Основни тип са којим се ради у *CPL*-у су апстрактни слогови. Пример 6 представља тип слога који описује податке о односима гена и протеина из базе података *GenPept*. Атрибут[22] `#feature` је скуп слогова који описују различите елементе. Сваки од њих, између осталог, има и атрибут `#anno`, који представља листу слогова. Пример 7 илуструје писање упита над таквим подацима. Конструкција `\x<-DB` означава да променљива `x` узима вредности из скупа `DB`. Слично томе, конструкција `\x<---lst` означава да променљива `x` узима вредности из листе `lst`.

Најзначајнији квалитет система *Kleisli* је у чињеници да успешно ради са разнородним изворима података. Један сложени упит се може односити на податке из више извора, чиме се апстрахује поступак интеграције података. Упитни језик је имплементиран кроз управљачке модуле за релационе базе података и друге изворе података.

Важно је нагласити да се у случају релационих база података делови упита пишу на *SQL*-у, да би се затим резултат упита на *SQL*-у даље обрађивао на *CPL*-у. Аутори система су се одлучили на такав приступ проблему повезивања са релационим базама података иако он ремети концепт пуне интеграције упитног и програмског језика. Штавише,

[22] У систему *Kleisli* се употребљава термин *поље*.

начин имплементације ремети и типизираност таквих упита, али су несумњиве предности употребе стандардизованог решења за рад са релационим базама података однеле превагу при дизајнирању тог аспекта система *Kleisli*.

```
(#uid: num,
 #title: string,
 #accession: string,
 #feature: { (
    #name: string,
    #start: num,
    #end: num,
    #anno: [ (
        #anno_name: string,
        #descr: string
        )]
    )}
)
```

Пример 6: Пример упита на *CPL*-у

```
let \DB' == { (#entry: x, #organism: a.#descr)
            | \x <- DB, \f <- x.#feature,
              \a <--- f.#anno, a.#anno_name = "organism" } in
let \ORG == { y.#organism | \y <- DB' }
in [ (#organism: z,
      #entries: { v.#entry | \v <- DB', v.#organism = z })
   | \z <--- [ u | \u <- ORG ] ];
```

Пример 7: Пример упита на *CPL*-у

Систем је развијан и примењиван првенствено у домену биоинформатичких анализа. У базе и колекције података са којима је систем успешно употребљаван спадају *Sybase, BLAST, ACeDB, OPM, ASN* и друге.

5.8. *HaskellDB*

Програмски језик *Haskell* је повезиван са базама података на више различитих начина. У контексту теме овог рада, најзначајнији резултат је *HaskellDB* [Leij1999]. У питању је библиотека за *Haskell* која омогућава записивање упита и других операција над базом података непосредно у програмском језику, без употребе упитног језика базе података.

Упит се записује као израз који израчунава листу одговарајућих редова. Могу се употребљавати операције рестрикције и пројекције, а подржане су и колонске функције. Једна од најважнијих карактеристика оваквог рада је строга типизираност, што омогућава да се евентуалне грешке у упитима распознају у фази превођења. У фази извршавања се упит израчунава тако што се прави и извршава одговарајући упит на упитном језику *SQL*. Приложени пример упита који користи библиотеку *HaskellDB* (Пример 8) је еквивалентан раније представљеном упиту на *SQL*-у (Пример 1).

Записивање упита на програмском језику представља вид пуне интеграције програмског и упитног језика. Такав приступ има и добре и лоше последице. У позитивне стране оваквог приступа се убраја:

- програмер не мора да учи и употребљава два различита језика;
- обезбеђена је строга статичка провера типова и

- могуће је на исти начин формално обрађивати цео програм, укључујући и делове који се односе на базу података.

Са друге стране, постоје и одређени проблеми:

- без обзира на чисто функционалну природу језика, неки елементи упита имају облик операционе, па чак и императивне нотације;
- иако се ради о једном језику, начин рада са упитима је значајно различит од осталих елемената језика, па се учење и употребе библиотеке у суштини не разликују од учења и употребе другог језика;
- у програмерској популацији је познавање релационих база података снажно корелисано са познавањем *SQL*-а, тако да оно што је наведено као прва позитивна страна, заправо има значајне негативне карактеристике и
- упити нису преносиви и не могу се правити и проверавати применом неких других алата.

```
upit = do { s <- table student
          ; i <- table ispit
          ; p <- table predmet
          ; restrict( s!indeks .==. i!indeks )
          ; restrict( p!id .==. i!id_predmeta )
          ; restrict( p!sifra .==. constant "P101" )
          ; project( indeks = s!indeks
                   , ime = s!ime
                   , prezime = s!ime
                   , dat_polaganja = i!dat.polaganja
                   , ocena = i!ocena )
          }
```

Пример 8: Израз на *Haskell*–у који помоћу библиотеке *HaskellDB* израчунава податке о студентима који су положили испит из предмета са шифром *P101*

Од осталих приступа проблему повезивања са базама података и програмског језика *Haskell* може се издвојити једно интересантно истраживање у коме је релациона база података апстрахована сложеним типовима података програмског језика [Silva2006]. Веома флексибилним средствима *Haskell*-а је направљен строго типизиран модел базе података. Моделиране су све основне операције над базом података, али и сложеније операције попут проверавања да ли је релација у некој од нормалних форми, композиције и декомпозиције релација. Метаподаци о бази података су представљени у облику типова, што омогућава да се помоћу таквог модела обављају статичке провере типова при раду са еквивалентном базом података под неким СУБП-ом.

5.9. *Tutorial D*

Дејт и Дарвин су у више радова разматрали могуће путеве развоја упитних језика [Darw1995, Darw2001, Date2007]. Као кључан принцип је претпостављено доследно бескомпромисно поштовање релационог модела података. Као резултат тих разматрања обликован је упитни језик *Tutorial D*, као апстрактан концептуални језик. Концептуална природа језика се огледа у томе да нису разматрани елементи језика који би се односили на повезивање са базом података, сесије, било какво приступање спољним ресурсима система (тј. било чему што није део базе података), управљање изузецима и друго. Језик

се ослања на релациону алгебру, која је допуњена до рачунски комплетног језика, уз пуну ортогоналност у односу на концепте објектно оријентисаног програмирања.

Постоји неколико решења која почивају на њиховом концепту. Један од значајнијих примера је релациони систем *Rel* [REL:2009]. Други пример је релациони систем *Dataphor* са упитним језиком *D4* [DCG:2009]. Систем *Dataphor* је имплементиран на платформи *.NET* и практично не представља праву базу података већ међуслој који као основу за управљање подацима употребљава неки други релациони СУБП. Оба примера обухватају одговарајуће имплементације варијанти језика *D*. Имплементације језика *D* се обично називају *Industrial D*, јер поред елемената језика *Tutorial D* имају и елементе неопходне за практичне примене које *Tutorial D* не обухвата.

Једну слабост оваквих решења представља управо одлика која је истовремено и њихов квалитет – тесна спрегнутост језика и базе података. Проблем је у томе што се на тај начин отежава повезивање са другим програмским језицима. *Dataphor* у значајној мери превазилази овај проблем, јер се у случају потребе за неким другим програмским језиком, тај језик може евентуално повезивати непосредно са конкретним СУБП-ом који се употребљава као основа за управљање подацима. Друга слабост је недовршеност, јер се ради о сасвим новим и још увек некомплетним решењима. Међутим, ако се на развоју настави темпом са којим се почело, оваква решења би могла да постану веома значајна и ван академских кругова.

Део II

Строго типизирани функционални програмски језици и базе података

6. Повезивање програмских језика и база података

6.1. Аспекти повезивања

Проблем повезивања програмских језика и база података је повезан како са особинама програмских језика и база података, тако и са карактеристикама окружења у коме се повезивање одвија. Сложеност односа програмског језика и базе података је последица великог броја различитих врста нивоа на којима се повезивање може остваривати, као и могућих смерова и начина повезивања.

Повезивање програмског језика и базе података се посматра, пре свега, на нивоу повезивања апликација, које се развијају на том програмском језику, и база података, које се из тих апликација употребљавају. Сагледавањем проблема са којима се суочавају развојни тимови, који имају потребе за коришћењем база података из програмских језика, и могућности којима располажу програмски језици и базе података, могу се уочити различите карактеристике односа језика и апликација са базама података, као и могућности за успостављање таквих односа.

У основне аспекте повезивања неког програмског језика и базе података спадају:

- архитектура повезивања;
- ниво апстрактности;
- смер повезивања;
- начин приступања подацима;
- модел трансакционог простора;
- типови података и
- начин похрањивања података сложених типова.

Ови аспекти повезивања одређују опште принципе, који се могу применити на произвољан програмски језик и произвољну базу података. У случају конкретних програмских језика и база података, овај скуп се може допунити одговарајућим специфичним аспектима.

У наредним одељцима се разматрају наведени аспекти повезивања програмских језика и база података. Посебна пажња се посвећује функционалним програмским језицима и релационим базама података. Питања која се односе на типове података се разматрају у поглављу 7.

6.2. Архитектура и ниво апстрактности

У савременим информационим системима се једна иста база података најчешће употребљава у већем броју програма који чине тај информациони систем. Да би било могуће да различите апликације употребљавају исту базу података, сам СУБП не сме бити интегрисан у неку појединачну апликацију. Због тога се уобичајено примењује *асоцијативно* повезивање апликације и базе података.

При развоју апликације се обично тежи да се бази података приступа коришћењем стандардних интерфејса, како би се имплементирани систем за управљање базама података или конкретна база података могли заменити другим системом или базом података. На тај начин се обезбеђује релативно виши ниво апстрактности повезивања.

Делови пословне логике, који се односе на читав информациони систем, уграђују се у базу података применом различитих концепата за аутоматизовање поступака на нивоу СУБП-а, као што су: страни кључеви, уведени услови, окидачи, серверске процедуре, проширивање упитног језика и слично. При томе се тежи да решења буду у што већој мери преносива, али то у пракси није увек у потпуности остварив. Проблем је у томе што сви расположиви системи за управљање релационим базама података, иако прилично доследно прате стандард *SQL*-а по питању самих упита, имају прегршт специфичности које ступају на сцену чим се са употребе упита пређе на напредније употребу система, а посебно у области аутоматизовања управљања подацима. Због тога напреднија употреба СУБП-а у највећем броју случајева има за последицу спуштање нивоа апстрактности и успостављање тешње спреге апликације (или, општије, решења информационог система) са конкретним СУБП-ом.

Ипак, иако се повезивање најчешће остварује тако да буде асоцијативно и релативно апстрактно, постоје и случајеви у којима се поступа сасвим другачије. У значајном броју пројеката се дешава да се развојни тим нађе у приликама којима уобичајена решења за рад са релационим базама података нису прилагођена . Тада се може бирати између сложене и често обимне или непримерене употребе уобичајених решења или једноставније и економски прихватљивије примене неких специфичних решења. Неки од узрока који могу довести до непрактичности уобичајених решења су:

- случајеви у којима долазе до изражаја слабости упитног језика *SQL* – на пример, евентуална навигациона природа података, велики број малих података различите структуре, потреба за објектном базом података и друго [Atki1989][23];
- сметње настале услед неких ограничења система за управљање база података – на пример, потреба за чувањем и коришћењем података који у основном облику нису подржани од стране система, потреба за похрањивањем и коришћењем програма и друго;

[23] У оваквим случајевима се релативно често употребљавају објектно оријентисане базе података.

- потреба за сложенијим механизмом трансакција, који обухвата не само базу података него и друге програмске ресурсе и
- друге специфичне околности.

У таквим случајевима се употребљавају, а често и развијају, решења која се одликују нешто широм интеграцијом алата за развој апликација и система за управљање базама података. Шира интеграција подразумева богатије односе алата и базе, али не и неопходно тесну спрегнутост развојних алата и базе. Одређивање уопштене „добре мере“ ширине спреге је веома незахвално. Штавише, услови су понекад довољно сложени да се таква решења развијају за једнократну примену.

Начин повезивања програмског језика и упитног језика има непосредан утицај на ниво и природу повезивања апликације, која се развија на програмском језику, и базе података, која се употребљава из те апликације. Према начину и степену интегрисања, повезивање програмског језика и упитног језика може бити остварено применом *пуне интеграције* или *концептуалне интеграције*.

6.2.1. Пуна интеграција програмског и упитног језика

Под повезивањем развојних алата и базе података се подразумева остваривање комуникације између делова програма писаних на неком програмском језику и система за управљање базама података. Како се приступање подацима у савременим системима за управљање базама података изводи употребом упитних језика, може се, без губљења општости, говорити о интегрисању програмских језика и упитних језика.

Један начин интегрисања је дефинисање неког језика који би истовремено представљао и програмски језик и упитни језик. Иако постоји више таквих покушаја, њихова примена у пракси је релативно уска. Један од разлога за слабо усвајање јединствених програмско-упитних језика је инертност индустрије, због чега свако ново решење има продужен период распрострањивања. Из разумљивих практичних разлога, а пре свега због цене и поузданости, индустрија тежи да примењује стандардизована или већ распрострањена решења за која постоји широка подршка, било од стране произвођача или других корисника.

Други важан разлог је у чињеници да сваки до сада направљен програмски језик има неке слабости, које чине да није погодан за примену у дословно сваком домену. Чак и језици који су декларативно општег домена, нису добро решење за многе специфичне проблеме, због тога што се начелна и практична примењивост могу значајно разликовати. Са друге стране, језик који се употребљава у раду са базом података мора да буде независан од домена, како би се иста база података могла користити у разнородним апликацијама које чине један информациони систем. Због тога везивање базе података за неки конкретан програмски језик непосредно смањује њену примењивост. Управо је независност и издвојеност *SQL*-а један од најзначајних чинилаца који су допринели широкој примени релационих база података.

Посебан проблем је у томе што значајан број тако интегрисаних језика не сарађује са системима за управљање базама података као независним компонентама, већ је сама имплементација језика тесно интегрисана и са имплементацијом система за управљање базама података. Било да се СУБП имплементира као део имплементације програмског језика, или се програмски језик имплементира као део имплементације СУБП-а, таквим

решењима се остварује двосмерна тесна спрега између апликативних компоненти и базе података, што представља негативан фактор при избору решења за развој информационих система.

И поред наглашених проблема, постоје неке примене у којима пуна интеграција извесно доноси специфичне квалитете. Карактеристичан пример таквих примена је употреба система *Kleisli* у домену интегрисања различитих извора података у области биоинформатичких истраживања. У таквим условима, када не постоји унифициран упитни језик који може читати податке из свих потребних извора, систем *Kleisli* је понудио квалитетно решење за постављање упита над разнородним подацима који потичу из више различитих извора података. Међутим, чак и тада, систем је ослоњен на *SQL* у мери у којој је то могуће (видети одељак 5.7).

6.2.2. Концептуална интеграција програмског и упитног језика

Алтернатива пуној интеграцији програмског и упитног језика је концептуална интеграција. Уместо потпуног интегрисања, у случају концептуалне интеграције програмски језик и упитни језик остају две независне целине, с тим да се прецизно одређују концепти њиховог повезивања, који су у основи независни од програмског језика и СУБП-а.

Једну врсту концептуалних интеграција представљају програмски интерфејси према базама података који се за конкретне програмске језике, или чак конкретне развојне алате, обликују тако да се различитим системима за управљање базама података приступа на исти начин. Оваква интеграција се може сматрати концептуалном само у погледу система за управљање базама података, док је програмски језик обично једнозначно одређен. Ограничавање слободе избора програмског језика у оваквим интеграцијама је последица различитости основних концепата језика, које онемогућавају доследну примену истих интерфејса у различитим језицима.

Функционални програмски језици се одликују далеко чистијом и једноставнијом семантиком него императивни језици. Због тога у случају функционалних програмских језика предузимање концептуалне интеграције представља нешто једноставнији подухват.

При предузимању концептуалне интеграције програмских језика и база података неопходно је суочити се са више проблема, који су, углавном, последица одређених ограничења на страни програмског језика или базе података. У основна ограничења имплементација на страни базе података спадају [Митић1995]:

- неадекватан систем типова;
- непостојање подршке за апстрактне типове података и
- некомплетан механизам за статичку проверу типова.

Са друге стране, у програмским језицима углавном нису подржане неке од уобичајених особина база података [Митић1995]:

- перзистентне структуре података;
- независност података од грешака у хардверу;

- конкурентност на нивоу података и
- механизми за контролу приступа, заштиту података, ауторизацију и сл.

Доследна концептуална интеграција би требало да омогући рад са свим елементима језика на исти начин – ортогонално и симетрично у сваком погледу. Употреба функционалних програмских језика при раду са базама података уводи неке специфичне проблеме, али доноси и низ нових квалитета. Основне особине функционалних програмских језика их чине погодним за писање програма у којима се употребљавају подаци из база података. Са друге стране, увођење концепта који би омогућио мењање података у бази података потенцијално нарушава принцип референцијалне транспарентности, што може да представља озбиљан проблем.

На пример, како у функционалним језицима функције имају исти статус као остали типови података, пуна примена похрањивања сложених података у бази података би требало да обухвати и могућност да се у бази података похрањују и делови програма, па и читави програми. Употреба елемената програмског језика у упитима би, у том случају, требало да обухвати и израчунавање програма и делова програма похрањених у бази података.

6.3. Смер повезивања

Уобичајено је да се повезивање система за управљање базама података са програмским језицима одвија у једном смеру. Програми најчешће имају улогу клијента који користи услуге СУБП-а ради приступања подацима похрањеним у базама података.

Са друге стране, програми или одговарајући програмски модули могу бити коришћени као сервери од стране СУБП-а, ради обављања одређених специфичних послова у области одржавања података или израчунавања. Примери програмских модула који имају улогу сервера су серверске процедуре (енгл. *stored procedure*) или кориснички дефинисане функције (енгл. *User Defined Function - UDF*). Такви модули се могу развијати на различитим програмским језицима, при чему се морају обезбедити одговарајући интерфејси за њихово покретање, како би их СУБП могао употребљавати. Сваки конкретан СУБП има специфичан интерфејс за покретање серверских програмских модула који мора бити поштован при њиховом развоју. Самим обликом интерфејса, али и неким другим специфичним карактеристикама конкретних система за управљање базама података се сужава избор програмских језика који се могу употребљавати у те сврхе.

Уколико се развија апликација која има потребе за уграђивањем одређеног програмског кода у саме наредбе *SQL*-а, које она употребљава, тада се развој програма који користе наредбе *SQL*-а и модула који се употребљавају у тим наредбама по правилу раздваја на потпуно одвојене развојне целине. На пример, ако је у програму неопходно употребити упит који употребљава неку специфичну функцију, тада се засебно и углавном независно развијају тај програм и имплементација такве функције. Често се при томе употребљавају различити програмски алати па и програмски језици.

Таква раздвојеност развоја има за последицу да се развоју кориснички дефинисаних функција приступа само у случају крајње неопходности, а да у већини случајева израчунавање остаје да се одвија на страни клијента. У таквим случајевима се од базе

података према клијенту (а често и од клијента према бази података) преносе много веће количине података него што је неопходно.

Проблем би се могао превазићи ако би у оквиру наредби упитног језика било допуштено коришћење сегмената програмског кода писаног на програмском језику на коме се развија клијентски програм. У том случају би се клијентски и серверски делови програма могли развијати као једна целина, чиме би се развој могао поједноставити. Наравно, ако би сложеност уграђених сегмената кода била повећана или њихова примена била понављана у више програма, они би се могли издвојити у посебне програмске библиотеке, али би се такве библиотеке могле употребљавати и у клијентским и у серверским деловима програма.

Аутору су позната само два покушаја имплементације сличног повезивања [Митић1995, Ora:2002], али је у оба случаја у питању само синтаксна олакшица, док се програмски код уграђен у упит извршава на страни клијента, пре или после извршавања упита у који је уграђен. У случају повезивања *Lisp*-а [Митић1995] ради се увек о извршавању кода на страни клијента пре позивања израчунавања упита, а у случају интерфејса *SQLJ* [Ora:2002] може да се експлицитно или имплицитно (на основу контекста) одреди да ли се код извршава пре или после упита или наредбе упитног језика.

6.4. Начин приступања подацима

6.4.1. Смер преношења података

Везу програмског језика и базе података је могуће остварити тако да буде подржано само читање података или и читање и мењање података у бази података. У случају повезивања функционалног програмског језика и базе података, може имати смисла да се функционалност ограничи само на читање података, како би интегрисано решење било слободно од бочних ефеката и у већој мери сагласно са принципима функционалног програмирања.

Када је реч о мењању података, начелно се могу разликовати случајеви додавања, брисања или мењања постојећих података. Свака од операција се може посматрати на посебан начин у контексту сложености података – тако, на пример, може да се подржи мењање података само у случају простих типова, а да се у случају сложених типова омогући само читање. На тај начин би се, можда, олакшала имплементација, али би се направили нови проблеми јер би се пореметила равноправност типова података. Посебно, увођење таквих разлика нема много смисла у случају функционалних програмских језика. Може олакшати имплементацију, али не доноси друге квалитете.

6.4.2. Статички и динамички SQL

У односу на тренутак извођења синтаксне и семантичке анализе упита и наредби, употреба конструкција језика *SQL* у неком програмском језику може бити *статичка* и *динамичка*.

Статичка употреба *SQL*-а подразумева да се и синтаксна и семантичка анализа обављају у тренутку превођења програма. Штавише, у тренутку превођења може да се направи и план извршавања, што обухвата и оптимизацију упита. Статички *SQL*

омогућава да се неке неисправности у програму уоче у фази превођења програма. Ипак, постоје и неке слабости оваквог приступа:

- У случају да се упити и наредбе односе на део базе података чији се садржај релативно динамично мења (на пример, табеле чија попуњеност у релативно кратком периоду варира од неколико редова до више милиона редова), план извршавања и одговарајуће оптимизације, који су направљени у време превођења програма, могу бити неприлагођени стању базе у тренутку извршавања. То може имати за последицу значајну неефикасност.
- Статички начин извршавања не омогућава извршавање наредби *SQL*-а које нису познате у тренутку превођења програма. Примери таквих случајева су интерактивни програми који омогућавају кориснику да уноси делове наредби или целе наредбе *SQL*-а.

Унапред преведени упити и наредбе *SQL*-а могу евентуално *постати* неисправни услед накнадних промена структуре базе података, до којих може доћи независно од конкретних наредби или апликације. Међутим, то се може догодити без обзира на начин превођења, па се не може сматрати слабошћу статичког *SQL*-а.

За разлику од статичког, *динамички* начин рада подразумева да се све у вези упита и наредби *SQL*-а, од синтаксне анализе па до извршавања, одвија у фази извршавања програма. Динамички приступ превазилази наведене слабости статичког приступа. Међутим, изостајање синтаксне и семантичке анализе у фази превођења има за последицу друге слабости:

- Евентуалне неисправности упита и наредби *SQL*-а се могу уочити искључиво у фази извршавања програма. То непосредно умањује поузданост програма.
- У фази превођења није могуће анализирати типове података, који се употребљавају или враћају у резултатима динамичких упита и наредби. Због тога се динамички *SQL* може имплементирати само уз примену неких генеричких конструкција, које не зависе од конкретних типова и омогућавају да се о типовима података стара програмер.

Имајући у виду да сваки од ова два приступа има и добре и лоше стране, пожељно је да постоји могућност употребе и статичког и динамичког начина рада.

6.4.3. Вредно и лењо читање резултата упита

Када се у оквиру програма израчунава неки упит, прву целину у поступку израчунавања представљају припрема и израчунавање резултата упита од стране СУБП-а. Другу целину представља преузимање појединачних редова резултата упита од СУБП-а и предавање програму на коришћење. При томе, преузимање редова може бити вредно и лењо.

Вредно израчунавање резултата упитне функције подразумева да се при сваком израчунавању упита, пре било какве употребе резултата, увек најпре прочитају сви редови који чине резултат упита. Такав приступ производи два основна проблема:

- Често се преносе и они редови резултата који нису потребни за израчунавање. Такође, праве се и одговарајуће репрезентације тих података у оквиру радне меморије која је на располагању програму. На тај начин се непотребно расипају меморијски, процесорски и мрежни ресурси.
- У зависности од нивоа изолованости трансакције[24], читањем редова „унапред", могу се ослободити неопходни катанци на прочитаним подацима и тиме омогућити њихово мењање од стране других процеса. Тако се може довести у питање изолованост, а тиме и поузданост и исправност трансакције.

Израчунавање резултата упита са лењом семантиком подразумева да се израчунавањем упита само започне поступак читања и преузимања резултата, а да се појединачни редови преузимају од базе података према потреби. Лењим читањем се решавају претходно наведени проблеми, али се производе нови, другачији. Могу се препознати два основна проблема који настају као последица одлагања читања редова:

- При употреби одложеног читања редова резултата упита, последњи редови тог резултата се неће преузети од СУБП-а све до пред сам крај израчунавања у коме се резултат упита употребљава. Ако је израчунавање сложено, па самим тим и дуготрајно, тада СУБП може дуго заузимати ресурсе система или чак (зависно од нивоа изолованости) и закључавати неке податке.
- Када се резултат упита употребљава у неком израчунавању, којим се изводе трансакције које могу утицати и на резултат управо тог упита, тада може бити неопходно, ради исправног тока трансакције, да се читав резултат упита ипак преузме од СУБП-а унапред, пре почетка мењања података.

На основу изложеног може се закључити да је пожељно да се програмски језик повезује са базом података тако да се резултати упита могу читати и вредно и лењо, а у зависности од конкретног случаја употребе.

6.4.4. Облик резултата упита

У императивним програмским језицима извршавање неког упита подразумева активно учешће у читању појединачних редова резултата, обично кроз експлицитно итеративно извршавање операције читања једног реда. У случају функционалних програмских језика такав приступ не одговара природи језика и често га није ни могуће остварити.

Природа функционалних програмских језика захтева да се резултат упита израчунава као једна целина. Због тога што је резултат у основи уређена колекција редова, природно је да се резултат у функционалном језику представи у облику листе, чији елементи одговарају прочитаним редовима. Имајући у виду чињеницу да је листа једна од основних структура података у функционалним програмским језицима, друга решења се практично повлаче у други план. Међутим, ни друга решења се не смеју

[24] На пример, у случају нивоа изолованости *Read Commited*, прочитан ред остаје закључан на нивоу СУБП-а све док се не прочита наредни ред. Читањем свих редова практично би се сви катанци постављени при читању упита (осим, евентуално, на последњем прочитаном реду) одмах и ослободили, тако да би евентуална каснија употреба тих података била сасвим независна од претходног читања, а тиме и непоуздана.

запоставити, јер листе у неким случајевима могу да буду неприхватљиве. Иако оне одговарају по својој структури, у зависности од специфичности конкретних функционалних програмских језика њихова семантика може бити проблематична.

У претходној дискусији о вредном и лењом читању изложен је закључак да је потребно да се читање редова резултата упита може одвијати и лењо и вредно, а у складу са конкретним околностима. У случају имплементације вредних програмских језика није увек могуће пружити непосредну подршку за лење листе. Један од начина имплементирања је да се уместо лење листе подржи апликативно решење које апстрахује рад са лењом листом редова помоћу одговарајућег скупа функција. На тај начин се рад са резултатима упита своди на императивни концепт курсора [Conn2004]. Са друге стране, иако може да буде далеко од тривијалног, вредно израчунавање резултата упита у лењим програмским језицима је ипак мањи проблем. Један начин обезбеђивања вредне семантике читања упита у лењом програмском језику је кроз имплементацију посебне функције која вредно израчунава све елементе дате листе.

Посматрано на концептуалном нивоу, закључак је да се за представљање резултата могу употребљавати апстраховане листе, при чему од природе језика зависи у којој ће мери бити апстраховани рад са њима и њихова потенцијално лења природа.

Сваком реду резултата упита одговара по један елемент листе (био он вредно прочитан или не). За представљање редова је најприхватљивије употребљавати податке неког структурног типа, који омогућава да се елементима структуре приступа по имену. Тада би сваки прочитан атрибут могао да се преслика у тачно један елемент структуре. Штавише, такво пресликавање би омогућило интуитивну употребу резултата упита, јер би оно пресликавало вредности и имена атрибута базе података у вредности и имена елемената структуре програмског језика.

6.5. Модел трансакционог простора

Интеграција програмског језика и базе података није потпуна ако се не односи и на саме механизме читања и мењања података. Једна од најважнијих карактеристика система за управљање базама података је постојање подршке за трансакције. Трансакциони механизми обезбеђују исправност података, како на нивоу појединачних мењаних података тако и на нивоу базе података као комплексне колекције међусобно зависних података. Са друге стране, у случају програмских језика се обично подразумева да су програми међусобно изоловани, тј. да користе податке у радној меморији који нису дељиви међу различитим програмима или различитим инстанцама истог програма.

Савремене апликације, а посебно информациони системи, у великој мери почивају на дељењу података од стране више корисника (тј. програма) истовремено. Штавише, потреба за дељењем података представља један од основних мотива за употребу база података које су независне од специфичних апликација. На пример, у случају Веб апликација, већи број појединачних програма, или различитих инстанци истог програма, може користити податке који се деле на нивоу сесије. Сама чињеница да се подаци деле наводи на размишљање о увођењу подршке за транскационе механизме на нивоу програмског језика помоћу којих се имплементирају такве апликације. Тренутно чак ни савремени програмски језици углавном немају такву подршку на нивоу самог језика.

Једна од основних карактеристика односа проблема приступа подацима и проблема интеграције програмског језика и базе података јесте постојање више различитих простора података. Простори података који се препознају у случају интеграције програмског језика и базе података су:

- простор података базе података – Чине га сви подаци похрањени у бази података. Њихово коришћење се одвија искључиво кроз трансакционе механизме базе података.
- локални простор привремених података програма – Овај простор података чине подаци у радној меморији рачунара, којима приступа једна инстанца програма током извршавања. Коришћење ових података се одвија непосредно, без подршке или употребе транскационих механизама.
- глобални простор трајних података апликације - Чине га подаци на нивоу апликације или сесије, које истовремено може употребљавати више програма или инстанци једне апликације. Овај простор података обухвата и систем датотека, уколико га апликација употребљава. Коришћење података се уобичајено одвија непосредно или кроз различите библиотеке (функција или објеката), без подршке за транскационе механизме у самом језику.

Обезбеђивање јединственог простора података за све трајне податке је потенцијално веома корисна димензија интегрисања програмског језика и базе података. Такав јединствен простор би представљао вид уједињавања простора података базе података и глобалног простора трајних података апликације. Кључни проблем је у томе да се обједињавање простора података може постићи искључиво у пару са обједињавањем односа према приступању подацима, тј. трансакцијама. Обједињени простор података би требало да представља јединствен трансакциони простор. Таквим обједињавањем трансакционог простора би се непосредно добило на поузданости програма.

Јединствен трансакциони простор подразумева подршку за јединствен трансакциони механизам који обухвата не само податке у бази података него и трајне податке који се употребљавају на нивоу апликације, а тиме и на нивоу програмског језика. У том контексту су од значаја све основне особине трансакција, али пре свега атомичност и изолованост.

Аспект изолованости се односи на начин међусобног изоловања како различитих делова апликације или инстанци њихове употребе, тако и различитих инстанци употребе базе података. При томе могу постојати разлике у приступу проблему изолованости у односу на читање и у односу на мењање података. У различитим случајевима ниво изолованости делова апликације може бити прилично различит, чему је неопходно посветити посебну пажњу.

Атомичност се односи на могућност да механизам трансакција у интегрисаном решењу обухвата непосредно не само податке у бази података већ и све или неке податке на нивоу програмског језика. Атомичност је непосредно повезана са остваривањем повезаности простора података базе података и простора података програма.

Услед различитог смисла појма трајности на нивоу базе података и инстанце апликације, питање трајности се решава различито. У размерама извршавања апликације трајност подразумева „трајање све док траје извршавање апликације као

целине“. Иако тако посматран проблем трајности нема глобалне карактеристике, као проблем трајности на нивоу базе података, они ипак имају значајних сличности, јер се у оба случаја трајни подаци могу употребљавати од стране више различитих програма који се временски не преклапају. Такође, у контексту промене података, свака потврђена промена мора опстати докле год само складиште за податке постоји – у једном случају база података, а у другом инстанца апликације.

Питање конзистентности је у домену програмера и тестирања програма, као и у домену аутоматских провера на нивоу базе података. Ту нема неких значајних специфичности у интегрисаном решењу. Евентуално има смисла разматрати неки облик декларативних услова интегритета података, који обухватају и податке базе података и податке из домена апликације. Проблем са таквим концептом је у томе што имплицира тесну спрегнутост базе података и апликације.

7. Улога типова у повезивању

7.1. Типови у базама података

Основни типови података су у савременим базама података подржани на релативно уједначен начин. Са друге стране, подршка за сложене типове је знатно неуједначенија. Основни типови података који се похрањују у базама података обухватају различите типове бројева и ниски, као и различите временске типове. Поред тога, на неки начин су подржани и различити облици бинарних садржаја. Бинарни садржаји представљају уопштене низове бајтова, који могу да садрже произвољне податке, слично као што је случај са датотекама. Такви подаци се похрањују у бази података али базе података не располажу неким напреднијим механизмима за обраду таквих података. Штавише, у самој бази података обично не постоје додатне информације о природи бинарних садржаја и начину њихове употребе, већ се обрада таквих података своди на читање или непосредно мењање дела бинарног садржаја.

Имајући у виду различите намене програмских језика и СУБП-а, природно је да постоје разлике у улози типова, па и начину њихове употребе. У случају база података, примарна улога типова је описивање домена података који се чувају у бази података. Информације о типовима података се употребљавају за статичко проверавање усаглашености дефинисаних односа међу подацима са типовима одговарајућих података, као и за динамичко проверавање исправности података који улазе у базу података.

Различите намене имају за последицу значајне разлике у одговарајућим системима типова. Разлике се највише огледају у области:

- нумеричких типова;
- типова ниски;
- структура и колекција података;
- функцијских типова и
- полиморфних типова.

У програмским језицима се уводи већи број различитих нумеричких типова ради управљања ефикасношћу израчунавања уз пружање потребног нивоа тачности података. Због тога што савремене рачунарске архитектуре омогућавају једнаку

ефикасност у раду са појединачним 8-битним и 32-битним целобројним подацима, један број савремених програмских језика не располаже већим бројем различитих целобројних типова (нпр *Java*, *C#* и други). Такође, како савремене рачунарске архитектуре подразумевају посебну рачунску јединицу за операције у запису са покретном запетом, која практично увек ради у истом (највишем) режиму тачности, често није неопходно ни постојање бољег броја различитих типова бројева записаних у облику са покретном запетом[25].

Са друге стране, у базама података се различитост у типовима уводи ради повећавања ефикасности складиштења и тачнијег одређивања опсега података. Због тога већи број различитих нумеричких типова података опстаје и у савременим базама података.

Слично је и у случају ниски. Ниске се у савременим програмским језицима углавном употребљавају на исти начин, без обзира на дужину. Са друге стране, ради успешнијег оптимизовања складиштења података, у базама података је веома значајно да дужина ниски буде што тачније предвидива, због чега постоји већи број одговарајућих типова. На пример, у случају система *IBM DB2*, постоје ниске фиксне дужине (`CHAR`), ниске променљиве дужине (`VARCHAR`), велике ниске (`LONG VARCHAR`) и веома велике ниске (`CLOB`), при чему је сваки од ових типова параметризован горњом границом дужине ниски [IBM:2008].

Највећа разлика је у односу на структуре и колекције података. Кроз подршку за различите структуре и колекције података у програмским језицима се омогућава да се имплементација програма ближе прилагоди специфичностима проблема који се решава. Са друге стране, базе података подразумевају једнобразан приступ представљању структура и колекција података, како би се независно од контекста оне увек могле користити на исти начин за претраживање или одржавање података. Управо је та једнобразност, поред формалног заснивања релационог модела података, један од основних разлога за широку примењивост концепта релационих база података.

У програмским језицима и потпрограми могу имати улогу предмета обраде. Таква улога је посебно значајна у функционалним програмским језицима, али је, у мањој мери, присутна и у већем броју императивних језика. У базама података често постоји могућност дефинисања програмског кода, који се употребљава на страни сервера чак и за релативно сложене обраде података. Међутим, такав програмски код се не понаша као садржај базе података, већ као њен структурни део. Због тога системи типова база података немају подршку за функцијске типове.

Базе података су намењене раду са конкретним подацима, који сами по себи немају апстрактну природу. Услед тога, системи типова база података не обухватају подршку за полиморфне типове. То се у пракси не препознаје као проблем, зато што сваки конкретан податак има и конкретан тип. Међутим, у контексту разматрања функцијских типова неизоставно је и разматрање полиморфних функцијских типова.

---

[25] Све наведено се односи, пре свега, на програмске језике општег домена. У одређеним областима је свакако неопходно искористити и најмањи простор за оптимизације. На пример, при писању управљачких модула за хардверске уређаје се често морају узимати у обзир све специфичности конкретних уређаја како би се постигао највиши степен перформанси.

7.2. Типови у вишејезичним окружењима

Да би се могли повезати програмски језик и база података, неопходно је на одговарајући начин повезати њихове системе типова. Такво повезивање је могуће остварити на три основна начина:

- пресликавање типова програмског језика у систем типова базе података;
- пресликавање типова базе података у систем типова програмског језика и
- пресликавањем типова програмског језика и базе података у неки независан систем типова.

У програмском језику се обично може препознати подсистем типова који углавном одговара систему типова базе података. Са друге стране, у бази података се обично може пронаћи (под)систем типова који одговара само врло уском скупу типова програмског језика.

Пресликавање типова програмског језика у систем типова базе података подразумева да се за што већи подскуп типова програмског језика препознају одговарајући типови базе података који им највише одговарају. Слично томе, пресликавање типова базе података у систем типова програмског језика подразумева да се за што већи подскуп типова базе података препознају одговарајући типови програмског језика који им највише одговарају.

Како повезивање обично захтева двосмерну размену података (и читање и писање), очигледно је да није довољно остварити само једно од наведена два пресликавања, већ да се она морају остваривати заједно. Идеалан случај би био када би једно од ових пресликавања било инверзно другом, али то у пракси није могуће остварити. Један од разлога за то су, на пример, ниске променљиве дужине – у бази података су типови ниски везани за њихову дужину, а у програмским језицима обично није тако, због чега се више типова базе података пресликава у један тип програмског језика.

Алтернатива је пресликавање система типова програмског језика базе података у неки трећи независан систем типова[26]. Независни системи типова се често употребљавају у окружењима која омогућавају писање програма на више различитих програмских језика или употребу библиотека писаних на различитим језицима. Систем типова, који је независан од конкретних програмских језика, омогућава формалније повезивање делова кода и на тај начин представља безбедносни мост између различитих програмских језика.

Развој независних система типова је текао упоредо са обликовањем језика за описивање апликативних интерфејса различитих облика дељеног кода. Циљ обликовања оваквих језика је да се омогући размена информација о деловима кода на начин који не би зависио од конкретне имплементације. Различити језици за описивање интерфејса омогућавају описивање библиотека делова кода (функција, класа и друго) и библиотека услуга (енгл. *service*).

---

[26] Овакви системи типова се често називају и *универзаланим* системима типова. Како већина корака у одговарајућем смеру нема за резултат дословно универзална решења, овде се употребљава термин *независни системи типова*.

Један од најутицајних корака у том смеру представља *IDL* (енгл. *Interface Description Language*[27]) [Gord2001]. Различите верзије овог језика за описивање интерфејса су развијане у складу са потребама конкретних система за повезивање независно развијаних делова кода, компоненти или услуга. У најзначајније се убрајају *OMG IDL*[28] [OMG:2002], *MIDL*[29] [MS.MIDL] и *WSDL*[30] [W3C:2007]. Једна од заједничких карактеристика система типова који одговарају наведеним језицима је да је тежња универзалности и флексибилности имала предност у односу на строгост, тако да сваки од њих подржава неку врсту *универзалног типа* који допушта практично било какве врсте података.

Специфичан облик независног система типова заузима веома значајно место у развоју тзв. међујезика, који омогућавају да се повезивање делова кода развијаних на различитим програмским језицима остварује тако што се сви ти делови кода преводе на исти међујезик, који се затим интерпретира или даље преводи до ивршног кода. Такав међујезик представља *IL* (енгл. *Intermediate Language*), на који се преводе сви програмски језици који су подржани од стране платформе *.NET* [MS.NET]. Виши ниво међусобне интеграције делова кода, у односу на архитектуре *CORBA* [OMG:2002] и *COM* [MS.COM], омогућава и виши ниво строгости при проверавању типова, као и неке додатне погодности [Gord2001].

7.3. Записивање података сложених типова у бази података

При разматрању проблема повезивања програмског језика и базе података један од најважнијих и најсложенијих проблема јесте записивање произвољних података програмског језика у оквиру базе података. Значај овог проблема је у томе што обезбеђивање подршке за размену података сложених типова и њихово записивање у

---

[27] Данас се као званичан назив употребљава *Interface Description Language*, мада је у првим верзијама пун назив овог језика био *Interface Definition Language*. У сваком случају, обично се реферише у облику скраћенице *IDL*, која није мењана.

[28] *OMG IDL* је језик за описивање интерфејса који се употребљава у оквиру архитектуре *CORBA* [OMG:2002] за повезивање „апликативних објеката", који могу бити независно развијани и извршавати се у једном истом или у различитим адресним просторима, укључујући и физички раздвојене рачунарске системе. *OMG IDL* је строго типизиран, у мери у којој то може бити једна спецификација интерфејса која тежи да буде *универзална*. Сваки објекат који се описује има свој тип, који чине назив типа, операције које може да изврши и њихови типови, као и променљиве којима располаже и њихови типови. Дефинисање (описивање) новог интерфејса је еквивалентно увођењу новог типа.

[29] *MIDL* је језик за описивање интефејса у оквиру архитектуре *COM* (енгл. *Component Object Model*), коју је развио *Microsoft* за примену на својим оперативним системима, првенствено за програмске језике *C++* и *Visual Basic* [MS.COM].

[30] Савремени приступ развоју сложених система почива на примени архитектура оријентисаних према услугама (енгл. *SOA – Service Oriented Architecture*). Један од најраспрострањенијих приступа је примена Веб услуга (енгл. *Web Service*). Име дугује двема основним сличностима са претходним Веб технологијама: употребљава исте комуникационе протоколе и исту начелну независност различитих чворова у мрежи. *WSDL* (енгл. *Web Service Description Language*) је језик за описивање интерфејса Веб услуга.

бази података доприноси повећавању изражајности и употребљивости интегрисаног решења. Тешкоће почивају, пре свега, на различитој природи система типова програмског и упитног језика. На разлике између програмских језика и база података по питању сложених типова је указано у одељку 7.1.

Под сложеним типовима података се у даљем тексту подразумевају различити типови података који су уобичајени у функционалним програмским језицима, а не спадају у просте типове. У такве типове спадају листе, торке, слогови, низови, функције и други сложени типови који се могу наћи у програмским језицима.

Релационе базе података су, пре свега, прилагођене раду са простим типовима. Неки од сложених типова се могу релативно једноставно похрањивати у бази података, али је увођење подршке за неке друге типове релативно сложено. Подршка за торке и слогове, као појединачне вредности у бази података, није елементарно обухваћена, али се може релативно једноставно обезбедити пресликавањем једног сложеног податка у више различитих атрибута. Највећи проблем представља омогућавање рада са функцијским типовима, који се интензивно употребљавају у функционалним програмским језицима.

Да би се тип резултата упита могао установити у фази превођења програма и провере типова у програмском језику, тј. пре извршавања програма и упита, неопходно је да постоји начин да се на основу текста упита установе типови података који се читају. То је могуће ако и само ако се тип прочитаних података установљава не на основу конкретних прочитаних вредности, или конкретних наведених услова, већ само на основу типова релација и атрибута из којих се подаци читају.

Све вредности једног атрибута морају да имају типове који су сагласни са декларисаним типом атрибута. Сагласност подразумева да те вредности морају припадати или том истом типу, или евентуално његовим подтиповима. Тиме се отвара једно ново питање: како обезбедити да вредности атрибута у бази података буду проверени сложени типови, ако сама база података нема подршку за те сложене типове? У наредним одељцима су продискутовани следећи могући путеви за решавање овог проблема:

1. проширивање СУБП-а подршком за сложене типове програмског језика;
2. употреба генеричких типова података у бази података (на пример, ниске), уз дефинисање неопходних мета-података и начина њихове употребе и
3. комбинован приступ.

7.3.1. Проширивање СУБП-а подршком за сложене типове података

Проширивање СУБП-а подршком за сложене типове података би представљало најбољи приступ проблему, по више критеријума. Између осталог, уграђивањем подршке за сложене типове података у сам СУБП би се омогућило да се из саме базе података, тј. из њене схеме, добијају информације о типовима података у атрибутима, као и за прочитане (или мењане) податаке. Тиме би се обезбедио највиши могући ниво поузданости, када су у питању провере типова података.

Међутим, остваривост овакве подршке је вишеструко проблематична. Оно што је пресудно је неопходност модификовања самог СУБП-а. Штавише, не једног, него сваког

понаособ. Да би то било остваривo, потребно је да се стекну одређени услови, а у међувремену, истраживање је потребно усмерити и на другу страну. Један од најозбиљнијих проблема је у томе што сваки од система за управљање базама података има неке специфичности у погледу подржаних типова података, које отежавају практичне аспекте оваквог приступа проблему. Штавише, имплементације различитих СУБП-ова почивају на различитим концептима и технологијама, због чега увођење опште подршке за сложене типове података у пракси представља веома озбиљан проблем.

Објектно релациона проширења релационих база података

Од средине 1990-их година почело се са уградњом објектно оријентисаних проширења у релационе базе података. Добијена решења се називају *објектно-релациона проширења* или чак и *објектно-релационе базе података*. Приступи различитих произвођача том проблему су се иницијално значајно разликовали. Стандардизација одговарајућих концепата је почела од стандарда *SQL*:1999 [ISO:1999], али и данас постоје значајне разлике између различитих производа [Melt2002].

Показало се да објектно-релациона проширења *SQL*–а доносе неке погодности за развој решења код којих се већи део пословне логике имплементира на самом систему за управљање базама података. Ипак, савремени трендови развоја софтвера су другачији, јер се показало да имплементација пословне логике у оквиру посебног слоја, који јесте ослоњен али не и сувише јако везан за слој података, доноси низ предности [Fowl2002].

Посматрано из угла аутора, највећа слабост представљених објектно оријентисаних проширења језика *SQL* је у самом концепту проширивања упитног језика до програмског језика. Тиме се значајно мења основа језика и још више нарушава и иначе недоследна примена релационог модела. Резултат представља нехомогену и неортогоналну комбинацију релационог и објектног језика. Колико год да је *SQL* квалитетан као упитни језик, у улози програмског језика је прилично незграпан и непрактичан. Један од основних проблема је у слабим могућностима повезивања објектно-релационог модела, на коме таква проширења почивају, са конкретним програмским језицима на којима се развијају апликације.

Објектно релационе базе података омогућавају дефинисање класа и хијерархија класа, као и чување објеката тако дефинисаних класа у бази података. Могуће је претраживање по класама или хијерархијама класа, као и постављање упита чији резултати обухватају објекте различитих класа. Међутим, чак и тако проширени скупови подржаних типова су и даље релативно уски, јер је квалитетно подржано пре свега моделирање података, а не и класа које представљају логику програма. Како је систем типова у функционалним програмским језицима још општији, јасно је да су сложени типови из функционалних програмских језика још даље од тога да буду подржани.

Уочене слабости постојећих објектно релационих база података не значе да таква проширења немају смисла, већ само да су постојећа решења углавном заснована на погрешним претпоставкама. Добар објектно-релациони СУБП мора почивати на доследном поштовању релационог модела и ортогоналном повезивању релационог и објектног модела [Darw1995, Darw2001, Date2007][31].

[31] О томе је било речи у одељку 5.9, на страни 72.

7.3.2. Употреба генеричких типова података у бази података

Један од могућих начина записивања било ког податка, било ког типа (па и функционалног), јесте записивање тог податка у облику изворног кода који га дефинише или израчунава. Да би се могао проверити тип податка, потребно је да се у систем за управљање базама података угради функције које израчунавају тип података датог кода, или да се уз сам код на одређен начин записује и његов тип.

Ако се функционалан програмски језик прошири механизмом који за било коју конкретну или апстрактну конструкцију израчунава (или дефинише) њен изворни код, онда се тиме стварају претпоставке за записивање сваког могућег појединачног податка у облику његовог изворног кода на конкретном језику. Алтернатива је да се уместо изворног кода израчунава неки други специфичан запис којим се једнозначно одређује било која конкретна вредност или конструкција функционалног програмског језика. Тиме се може постићи боља заштита кода (у контексту ауторских права и сл.) али се смањује употребљивост таквог садржаја базе података, јер се подаци могу употребљавати само у контексту програмског језика који зна да их прочита, док се у случају записивања изворног кода они могу употребљавати и непосредно од стране корисника базе података, у нешто блаже ограниченом обиму.

Добра страна представљеног приступа је што није неопходно мењати сам СУБП, док је недостатак то што се провера типова не одвија у оквирима СУБП-а, чиме се умањује поузданост саме провере. И у овом случају је тип *SQL* упита неопходно експлицитно проверавати при превођењу програма. Тип се може провераваоти у контексту распложивих мета података, али без могућности да се поступак проверавања у значајној мери ослони на саму базу података. У упитима на *SQL*-у се подаци, чији се тип проверава на овај начин, могу употребљавати само у облику изворних кодова програма, тј. као ниске које евентуално могу да се обрађују на одређени начин.

7.3.3. Комбинован приступ

Алтернатива решавању проблема искључиво на једној страни би било остваривање комбинованог приступа. Комбиновани приступ подразумева да се у бази података дефинишу нови типови података, у складу са могућностима конкретног СУБП-а, али да се за физичко представљање података употребљава неки генерички облик, који би се могао у пракси употребљавати од стране програмског језика. Пресликавање тако дефинисаних типова у бази података у одговарајуће типове у функционалном програмском језику се мора одредити одговарајућим мета подацима.

У случају комбинованог приступа, проверавање типова података може да се одвија једним делом на страни СУБП-а, а другим делом на страни програмског језика. Тиме би се практично задржале неке добре стране како строге типизираности тако и генеричких типова података, али би се истовремено омогућило превазилажење најважнијих проблема које носи њихова пуна примена.

7.4. Типови и начини приступања подацима

Начини приступања подацима су разматрани у контексту уопштеног приступа овом проблему, у одељку 6.4 Употреба строго типизираних функционалних програмских језика доноси одређене специфичности, којима је овде посвећена пажња.

У претходној уопштеној дискусији је истакнуто да је за представљање резултата упита најбоље употребљавати листу редова, при чему се сваки ред представља неким структурним типом који омогућава да се елементима структуре приступа по имену.

У контексту типова, закључак се може проширити захтевом да типови елемената који чине тај структурни тип буду међусобно независни, тако да имену, типу и вредности сваког од прочитаних атрибута одговарају име, тип и вредност по једног елемента структуре. На тај начин би се стекли услови за типизирано повезивање програма и базе података.

Генеричка употреба резултата упита

Постоје случајеви у којима типизиран приступ може да донесе одређена релативно значајна ограничења. У развоју софтвера је често потребно написати делове кода који омогућавају обраду произвољних упита или података, тако што се односе генерички према одређеним деловима посла. У контексту повезивања програмског језика и база података генерички приступ би био неопходан, на пример, ради писања уопштене функције која прави извештај о резултатима извршавања упита над базом података за било који конкретан упит. Пример таквих генеричких елемената кода представља механизам рефлексија у програмском језику *Java* [Arno1996].

Генерички приступ доноси значајну флексибилност, али непосредно доводи у питање типизираност. Због тога захтева повећану опрезност корисника.

Нетипизирани резултати упита, који омогућавају генеричку употребу, се могу начелно обезбедити на два начина:

- омогућавањем постављања слабо типизираних упита или
- додавањем примитивне функције која пресликава типизирани структурни тип у одговарајући слабо типизиран облик.

„Нетипизирано“ представљање реда резултата упита је могуће остварити на флексибилан начин помоћу каталога, коме су и кључеви и вредности ниске. Кључеви би одговарали називима, а вредности управо вредностима атрибута.

Ради обезбеђивања пуне флексибилности у генеричкој примени резултата упита пожељно је додати и примитивну функцију којом би се сви кључеви каталога могли издвојити у листу, или неку другу функцију којом би се омогућила анализа постојећих атрибута.

8. Принципи и концепти повезивања

8.1. Добре особине програмских језика

Интеграција програмског језика и база података подразумева одређене промене или надградње у програмском језику и бази података. Да таквим променама не би биле доведене у питање постојеће карактеристике програмског и упитног језика, потребно је укључити у разматрање проблема и тзв. *добре особине* програмских језика.

У савременој научној литератури у области програмских језика (нпр. [Yaof2005]), као неке од најважнијих особина које неки програмски језик чине погодним за примену препознају се:

- *Упштеност* – Пожељно је да домен у коме се језик може применити буде што шири, а конструкције језика што слободније и независније од специфичне примене;
- *Ортогоналност* – Конструкције језика је потребно да буду међусобно независне и неограничавајуће, тако да се могу слободно комбиновати, а да при томе не буде неочекиваних резултата, који не прате општи концепт језика;
- *Поузданост* – Језик мора омогућити писање програма који су поуздани;
- *Одрживост кода* – Синтакса и стил писања који сам језик имплицира морају бити такви да помажу читљивости кода и његовом лаком одржавању. Велики утицај на одрживост кода има како свака конструкција језика понаособ, тако и међусобна ортогоналност тих конструкција;
- *Ефикасност* – Програмски језик не сме међу својим основним особинама обухватити неке претпоставке које би представљале концептуалне препреке ефикасном извршавању програма. Елементи који су потенцијално скупи морају бити лако препознатљиви;
- *Једноставност* – Једноставност се односи како на заокруженост и целовитост основних концепата и конструкција језика, тако и на њихову укупну минималност и одсуство међусобног преклапања;

- *Машинска* независност – Ова особина одражава у којој мери је семантика програмског језика независна од архитектуре и карактеристика рачунарског система на коме се језик може имплементирати;
- *Примењивост* – Елементи језика морају бити разумљиво и у потпуности дефинисани, како би се могли имплементирати на прихватљив начин;
- *Проширивост* – Могућност да језик употребљава различите библиотеке кода, да се допуњава новим елементима, проширује или дограђује, представља једну од основних претпоставки за његову успешну и трајнију примену;
- *Изражајност* – Могућност да се опишу сложени алгоритми и структуре података је неопходна за озбиљну примену и
- *Утицајност* – Ако је неки језик утицао на развој других програмских језика, али и ако је попримио неке особине већ раширених програмских језика, тада ће његово учење бити једноставније и почетак примене једноставнији.

У контексту повезивања програмског језика и језика база података, било да се повезивање одвија на апстрактном нивоу или на унапред изабраним језицима, на неке од ових особина се не може непосредно утицати. Са друге стране, неке особине добијају додатни значај, а пре свега уопштеност и ортогоналност.

Такође, премда је поузданост незаобилазна у разматрању свих фаза развоја софтвера, она посебно долази до изражаја у домену база података и различитим аспектима њиховог рада или њихове употребе. Чак и минимално угрожавање принципа поузданости довело би у питање смисао повезивања са базама података.

8.2. Принципи

На основу анализе проблема повезивања програмских језика и база података и увида у најважније карактеристике програмских језика, препознати су и обликовани основни принципи на којима је потребно да почива повезивање програмског језика и базе података. Наведени скуп принципа је могуће и проширити, али се испоставило да остали потенцијални кандидати који су разматрани током овог истраживања могу да се изведу из усвојених принципа.

Основни принципи на којима би требало заснивати интегрисање програмских језика и база података су:

- апстрактност;
- ортогоналност и
- симетричност.

Принцип 1 – Апстрактност

Концепти на којима се заснива повезивање програмских језика и база података се морају обликовати уопштено, тако да се без угрожавања основног смисла тако изражених концепата, они могу имплементирати при повезивању различитих програмских језика и база података.

Значај принципа апстрактности следи непосредно из значаја који има уопштеност језика. Висок ниво апстрактности концепата повезивања има за непосредну последицу низак ниво спрегнутости програмског језика и базе података, а затим и флексибилност добијеног решења.

Принцип 2 – Ортогоналност

Повезивањем се не смеју доводити у питање основне особине програмског и упитног језика, нити смањивати примењивост ових језика. Потребно је да се сви елементи ових језика могу употребљавати у што већој мери на исти начин као и пре повезивања.

Принцип ортогоналности истиче како је поред значаја ортогоналности сваког од језика важно да и сами концепти повезивања и конструкције на којима се повезивање заснива буду ортогонални у односу на оба језика. Широка примена овог принципа се односи како на конструкције оба језика, тако и на типове података.

Принцип 3 – Симетричност

Повезивање је потребно остварити у оба смера, уз што већу симетричност концепата и равноправност језика.

Принцип симетричности представља основу за флексибилност и двосмерност односа између програмског језика и базе података. Један од основних аспеката симетричности је да се поред употребе језика база података у програмском језику, омогући и употреба израза програмског језика у језику базе података. Последица принципа симетричности и ортогоналности би требало да буде и остваривање одређене равноправност у односу на програмски код и податке оба језика.

Наведени принципи повезивања програмских језика и база података остављају више последица на препознате добре особине програмских језика. На основу принципа апстрактности и ортогоналности следи да би интеграција требало да почива на малом броју што чистијих концепата, који не нарушавају природу програмског и упитног језика. Тиме се имплицира очување *једноставности* и *одрживости кода*. Стварају се претпоставке и да се не доведу у питање друге важне особине језика: *примењивост*, *машинска независност*, *проширивост* и *изражајност*. Ако се узме у обзир почетно стање, пре интеграције, јасно је да би свака неортогонална измена нарушила неке од постојећих особина језика а тиме и посредно довела у питање компатибилност кода, као и неке од добрих особина језика.

8.3. Концепти

Ако се при повезивању програмског језика и СУБП-а у одређеним синтаксним конструкцијама употребљавају елементи и програмског језика и језика СУБП-а, онда се у таквим случајевима, ради њиховог јасног разликовања, програмски језик назива матични језик (енгл. *host language*), а језик СУБП-а се назива упитни језик.

Ако се у изразу упитног језика употребљавају имена дефинисана у деловима кода на матичном језику, таква имена се у изразима упитног језика називају матична имена. Ако матична имена представљају променљиве или формалне аргументе потпрограма матичног језика, тада се називају и матичне променљиве (енгл. *host variables*). У

функционалним програмским језицима не постоје променљиве у пуном смислу те речи. Ипак, у складу са претходним, за матична имена која представљају формалне аргументе функције у даљем тексту се употребљава термин *матична променљива*.

Запис израза упитног језика у оквиру матичног језика, који само чита садржај базе података, назива се упитни израз. Израз упитног језика који мења (или потенцијално мења) садржај или структуру базе података назива се упитна наредба. Упитни израз је типизиран ако се као резултат израчунава строго типизирана колекција података. Упитни израз је слабо типизиран ако се као резултат израчунава колекција података неког генеричког типа, који не зависи од података који се читају.

Примена усвојених принципа води стварању концепата чијим се остваривањем постиже интегрисање програмских језика и база података. Један број концепата је специфичан за строго типизиране програмске језике, док су остали концепти примењиви на практично све програмске језике. Сви наведени концепти су међусобно ортогонални, па је у случају да се неки од њих не може применити и даље могуће применити остале концепте.

Уз већину концепата су у облику *потконцепата* наведене неке последице или препоручени начини имплементирања.

Концепт 1 – Употреба упитног језика у програмском језику

Приступање садржају базе података је потребно остваривати конструкцијама упитног језика, које се на одговарајући начин наводе у оквиру кода на програмском језику.

Концепт употребе упитног језика у програмском језику пружа флексибилност и ортогоналност у највећој могућој мери. Доследна примена концепта подразумева могућност употребе свих исправних конструкција упитног језика у оквиру програма на програмском језику.

Већи број постојећих решења проблема повезивања почива на примени овог концепта. Решење које га најдоследније примењује је угњеждени *SQL*. Концепт употребе упитног језика у програмском језику је у значајној мери примењен и у многим другим решењима (на пример *ODBC, JDBC*). У случају угњежденог *SQL*-а се упитни изрази и упитне наредбе употребљавају у програмском коду уз коришћење специфичних синтаксних конструкција, којима се издвајају од „обичног" програмског кода. У случају *ODBC*-а и *JDBC*-а се изрази упитног језика наводе у облику ниски.

Када су у питању функционални програмски језици, без значајног губљења општости, а у складу са природом језика, могуће је ограничити употребу упитних израза и наредби на одговарајуће специфичне врсте функција.

Потконцепт 1а – Упитне функције

Упитни изрази се употребљавају у оквиру посебне врсте потпрограма – упитних функција.

Потконцепт 1б – Акционе функције

Упитне наредбе се употребљавају искључиво у оквиру посебне врсте потпрограма – акционих функција.

Јасним разграничавањем упитних израза и наредби од конструкција матичног језика се у пуној мери остварује ортогонално повезивање.

Концепт 2 – Употреба програмског језика у упитном језику

У оквиру упитних израза се могу употребљавати изрази програмског језика. Такви изрази се називају матични изрази. Матични израз се израчунава на серверу СУБП-а, у току израчунавања упитног израза у коме је наведен.

Пуна примена принципа ортогоналности и симетричности при повезивању програмског језика и базе података имплицира да није довољно омогућити да се конструкције упитног језика употребљавају у коду на програмском језику, већ је потребно омогућити и да се конструкције програмског језика могу употребљавати у оквиру кода који је написан на упитном језику. У постојећим решењима је таква могућност сведена на минимум кроз концепт матичних променљивих у оквиру уграђеног *SQL*-а. Принципи ортогоналности и симетричности налажу уопштавање концепта матичних променљивих, увођењем новог и далеко општијег концепта матичних израза.

Матични израз представља израз матичног језика записан у оквиру израза упитног језика. Основна идеја је да се као матични израз може навести сваки исправан израз програмског језика. Такав израз може да се без ограничења употребљава у упиту (који је на упитном језику), све док тип израза одговара контексту у коме се употребљава. Матични израз се може употребити на сваком месту у наредбама упитног језика где синтакса омогућава да се употреби нека скаларна вредност, листа скаларних вредности (тј. колона) или табела.

Као што је већ наглашено, матични изрази се израчунавају на серверу, у оквиру израчунавања резултата упита. Ако семантика конкретних матичних израза то допушта, израчунавање матичног израза или дела матичног израза се ради оптимизације може изместити тако да се одвија на страни клијента, пре или после израчунавања упитног израза.

Концепт 3 – Строга типизираност повезивања

На сваком месту где постоји размена података између кода на програмском језику и кода на упитном језику, потребно је омогућити да та размена података буде строго типизирана.

Строга типизираност елемената повезивања обезбеђује да и у условима повезивања са базом података строго типизиран програмски језик очува све оне позитивне карактеристике које следе из строге типизираности самог језика. На тај начин типизираност пружа веома важну димензију поузданости.

Строго типизирано повезивање има смисла чак и када се не ради о строго типизираном програмском језику, мада у таквим случајевима може да имплицира одређене промене у начину писања програма, што може да представља одређено нарушавање принципа ортогоналности.

Строга типизираност је углавном присутна у решењима која почивају на примени статичког начина повезивања са базама података (на пример, уграђени *SQL*), али не и у случају динамичког повезивања (на пример, *ODBC* и *JDBC*).

Потконцепт 3а – Строга типизираност повезивања – Генерички приступ

Пожељно је омогућити читање података из базе података и у неком генеричком облику, који не зависи од конкретних типова података који се читају.

Већ је указано на значај генеричког приступа и указано на основне могућности имплементирања (у одељку 7.4). Иако такав приступ не би требало да буде примаран, постоје случајеви у којима је неопходан, због чега је пожељно да буде подржан.

Концепт 4 – Јединствен трансакциони простор

Све промене трајних података се одвијају у оквиру јединственог трансакционог простора и кроз обједињен трансакциони механизам.

Концепт јединственог трансакционог простора је последица придржавања принципа симетричности у односу на трајне податке програмског језика и базе података. О мотивацији за увођење јединственог трансакционог простора је било речи у одељку 6.5.

У случају функционалних програмских језика примарно питање је да ли су трансакције уопште подржане. Чисто функционални програмски језици имају концептуални проблем у вези са било каквим променама података, па и са трансакцијама. У случају функционалних програмских језика који омогућавају бочне ефекте, могуће је прецизније одређивање овог концепта без значајног смањења изражајности језика.

Овај принцип није у потпуности остварив у општем случају, пре свега из разлога што највећи број програмских језика уопште не располаже транскационим механизмима. Ипак, његов значај налаже сагледавање услова за његову примену и у таквим случајевима.

Потконцепт 4а – Јединствен трансакциони простор – Локализација бочних ефеката

Сви бочни ефекти морају бити локализовани и јасно уочљиви.

Експлицитним локализовањем свих бочних ефеката помоћу посебних конструкција програмског језика се омогућава раздвајање делова кода који садрже бочне ефекте од делова кода који су чисто функционални. Тиме се стварају предуслови да се значајан део кода може третирати као чисто функционалан, како при пројектовању и писању, тако и при анализи или чак имплицитном верификовању или паралелном извршавању.

Потконцепт 4б – Јединствен трансакциони простор – Трансакционе функције

Трансакције се описују у облику посебне врсте функција – трансакционих функција. Трансакциони механизми се примењују имплицитно при израчунавању трансакционих функција.

Концепт трансакције је један од основних концепата у системима за управљање базама података [Date2003]. Најприроднији начин имплементације овог концепта у функционалним програмским језицима је увођење концепта трансакционих функција.

Трансакционе функције су атомичне целине кода које могу да производе бочне ефекте. Имплементацијом трансакционих функција са припадајућим подсистемом за

трансакције се истовремено постиже локализавање бочних ефеката и обезбеђивање јединственог трансакционог простора. Локализовање бочних ефеката се остварује ограничавањем могућности употребе операција које могу мењати податке искључиво на трансакционе функције.

Семантика трансакционе функције подразумева имплицитно започињање трансакције при започињању израчунавања функције и имплицитно завршавање трансакције при завршавању израчунавања функције. При завршавању трансакције она се потврђује или поништава, што зависи од тока и резултата израчунавања тела функције, а у складу са правилима конкретног програмског језика.

Уколико израчунавање трансакционе функције представља (непосредно или посредно) део израчунавања неке друге трансакције, онда се при њеном израчунавању не започиње (и не завршава) нова трансакција, већ се функција понаша као саставни део шире трансакције.

Потконцепт 4в – Јединствен трансакциони простор – Акционе функције

Акционе функције су уопштење трансакционих функција које не подразумева постојање трансакционих механизама. Акционе функције се могу употребљавати искључиво у оквиру трансакционих функција, било непосредно или посредно.

Раније је већ наведен концепт према коме се упитне наредбе употребљавају у оквиру акционих функција (1б). Може се поставити питање да ли трансакционе и такве акционе функције могу да представљају један исти појам? Заједничко за ове две врсте функција је локализовање бочних ефеката и могућност употребе упитних наредби.

Основна мотивација за задржавање два различита појма је у потреби да се чињеници да је у практичном раду при имплементацији трансакција потребно писати помоћне функције са бочним ефектима, које не могу да представљају самосталне трансакције јер не задовољавају услов конзистентности. Акционе функције, које не обухватају трансакционе механизме, представљају средство за непосредно решавање таквих проблема.

Потконцепт 4г – Јединствен трансакциони простор – Израчунавање упитне функције

Ако се упитна функција израчунава посредно или непосредно у оквиру израчунавања неке шире трансакције (трансакционе функције или друге упитне функције), онда је њено израчунавање обухваћено том широм трансакцијом. У супротном, ако израчунавање упитне функције није обухваћено израчунавањем неке шире трансакције, имплицитно се започиње нова трансакција, која обухвата израчунавање резултата упита и имплицитно се завршава по завршетку читања.

Овакво понашање може да се изведе из других концепата, али се овде наводи у облику потконцепта због тога што има велики значај.

Потконцепт 4д – Јединствен трансакциони простор – Ниво изолованости трансакције

Свака трансакција може да има посебан ниво изолованости, који се опционо наводи у оквиру дефиниције трансакционе функције.

Конкретна синтакса за навођење нивоа изолованости трансакције зависи од специфичности конкретних језика.

Потконцепт 4ђ – Јединствен трансакциони простор – Ниво изолованости упита

Ако се при израчунавању упитног израза започиње имплицитна трансакција, тада она има ниво изолованости који је наведен за дати упит. Ако се упитни израз рачуна у оквиру израчунавања неке шире трансакције, онда се употребљава ниво изолованости који важи за ту ширу трансакцију.

Конкретна синтакса за навођење нивоа изолованости упита зависи од специфичности конкретних језика. У случају функционалних програмских језика може се наводити у оквиру дефиниције упитне функције.

Потконцепт 4е – Јединствен трансакциони простор – Матични изрази

Ако матичн израз производи бочне ефекте, они морају да буду у оквиру истог трансакционог простора и исте трансакције као и упитни израз у коме се тај матични израз налази.

Мотивација за овакво правило проистиче на основу принципа ортогоналности из представљених концепата матичних израза и јединственог простора. И ово правило би могло да се сврста у последице изведене из других концепата, али је због његовог великог значаја ипак наведено експлицитно у виду потконцепта.

Концепт 5 – Ортогоналност у односу на типове података

У бази података се могу записивати подаци сваког од типова које подржава програмски језик. У програмском језику се могу употребљавати подаци свих типова које пофржава база података.

Један од најважнијих аспеката повезивања програмског језика и базе података јесте ортогоналност у односу на типове података. Наравно, овај концепт има посебну тежину у пару са концептом строго типизираног повезивања.

Потконцепт 5а – Програми у бази података

У сложене типове који се могу записивати у бази података спадају и функцијски типови.

Омогућавањем записивања података произвољног типа у бази података практично се омогућава и записивање програмског кода у бази података. Ипак, овај потконцепт се наводи и експлицитно, јер његова тежина није иста у односу на све програмске језике. Значај је посебно велики за функционалне програмске језике, а пре свега за строго типизиране функционалне програмске језике.

Потконцепт 5б – База података као библиотека кода

База података може да има улогу библиотеке програмског кода.

Корак даље у смеру записивања програмског кода у бази података представља проширивање програмског језика конструкцијама које омогућавају коришћење кода похрањеног у бази података на исти начин као што се употребљавају датотечне библиотеке функција. Основна разлика у односу на уопштено похрањивање

програмског кода у бази података је у потреби да једна библиотека обухвата дефиниције различитих типова.

Концепт 6 – Ортогоналност у односу на начин повезивања

Имплементација повезивања не сме да умањи општост и слободу у погледу начина остваривања повезивања.

У контексту начина повезивања овај концепт се односи пре свега на начин припремања и израчунавања упита (статички и динамички *SQL*), типизираност и семантику резултата упита (вредно и лењо читање).

Потконцепт 6а – Статички и динамички SQL

Потребно је омогућити и статички и динамички начин рада са базом података.

Претходно је наглашен значај обезбеђивања оба начина рада (одељак 6.4.2). Овај вид ортогоналности има посебан значај ако се имају у виду типови података и строга типизираност.

Потконцепт 6б – Строго и слабо типизирани резултати упита

И поред свих предности строго типизираног начина рада, потребно је омогућити и слабо типизиран начин рада са базом података.

Мотивација за овакав концепт је наведена у одељку 7.4. Суштина је у томе да строго типизиран начин рада не може да се сложи са динамичким радом и генеричком употребом података из базе података. У таквим случајевима се мора радити или нетипизирано или слабо типизирано. Због тога што у контексту овог рада нетипизирано решење није прихватљиво, потребно је да се обезбеди подршка за слабо типизиран начин рада.

Потконцепт 6в – Вредно и лењо читање

Потребно је омогућити и вредно и лењо читање података из базе података.

Значај оваквог концепта је објашњени у одељку 6.4.3, а могући путеви имплементирања у одељку 6.4.4.

9. Повезивање и провера типова

9.1. Провера типова при повезивању

Специфичности проверавања и распознавања типова у околностима повезивања програмског језика и база података се односе на проверавања сагласности типова података који се размењују између програмског језика и упитног језика. Издвајају се два кључна места на којима се таква размена података одвија. Једно место су упитни изрази и наредбе у програмском језику, а друго су матични изрази у упитном језику.

У овом поглављу су наведени алгоритми за проверавање сагласности типова за упитне и матичне изразе. Претпоставља се да се проверавање типова одвија статички, у фази превођења, у складу са строгом типизираношћу. Евентуално прилагођавање случају динамичког проверавања типова не захтева велике измене у алгоритмима. Такође, без значајног губитка општости, претпоставља се да су упитни изрази увек у оквиру одговарајућих упитних, акционих или трансакционих функција.

При распознавању типова података који се размењују између базе података и програмског језика постоји више елемената који се могу апстраховати. Ипак, када је у питању размена информација о типовима сложених података сачуваних у бази података, поступак је тесно везан за специфичности имплементације података сложених типова. Због тога се у наредним алгоритмима кораци, који се односе на размену таквих информација између СУБП-а и преводиоца, реферишу без икаквих детаља. То уједно представља и основну недореченост наведених алгоритама. Специфичне имплементације прилагођене програмском језику *Wafl* су наведене у поглављу 12.

9.2. Распознавање типа упитне функције

Строго типизирано повезивање програмског језика и базе података захтева сарадњу подсистема за распознавање типова програмског језика и одговарајуће компоненте СУБП-а. Додатну сложеност уводи подршка за сложене типове у бази података. Алгоритам распознавања типова упитних, акционих и трансакционих функција почива на распознавању типова по фазама и међусобној размени информација о типовима прочитаних атрибута између СУБП-а и програмског језика.

У случају упита у којима се употребљавају матичне променљиве, потребно је посебно узети у обзир и сваку од употреба матичних променљивих у изразу. Посебно, због

начина размењивања података и провера, потребно је раздвојити евентуалне различите употребе исте матичне променљиве, па затим проверити сагласност добијених информација. Наравно, ако се у упиту не користе матичне променљиве, онда алгоритам има нешто једноставнију структуру.

Алгоритам 5 је намењен за распознавање типа упитне функције. Због природе наредби упитног језика, само у случају упитних функција је потребно проверавати тип резултата, па је проверавање акционих и трансакционих функција нешто једноставније. Алгоритам је углавном универзалан и може се применити на било који програмски језик и базу података.

1. Ако упитна функција има аргументе, онда:
   1.1 Свако појављивање матичне променљиве у тексту упита се замењује јединственом ознаком матичног параметра, при чему се прави таблица пресликавања матичних параметара у одговарајуће параметре;
2. Управљачки модул тражи од СУБП-а да препозна тип резултата упита, као и да на основу места на коме се употребљавају распозна неопходне типове матичних параметара;
3. Ако СУБП успе да препозна све тражене типове, онда:
   3.1 Ако међу типовима које је СУБП препознао постоји неки од специфичних типова програмског језика, тада се сваки такав тип замењује одговарајућим типом програмског језика;
   3.2 Ако се разликују типови матичних параметара који се односе на једну исту матичну променљиву:
      3.2.1 Покушава се њихова унификација;
      3.2.2 Ако унификација не успе, значи да постоје неисправности у упиту:
         3.2.2.1. Прекида се провера типа;
         3.2.2.2. Преводилац се обавештава о проблему;
   3.3 Од препознатих типова матичних променљивих и резултата се прави одговарајући функцијски тип и прослеђује се преводиоцу као тип упитне функције, како би био искоришћен у даљем проверавању типова у програму;
4. Иначе, значи да постоје неисправности у упиту:
   4.1 Прекида се провера типа;
   4.2 Преводилац се обавештава о проблему.

Алгоритам 5: Препознавање типа упитне функције (без матичних израза)

9.3. Распознавање типа матичног израза

У претходним одељцима је описано распознавање типа упитних функција у којима се не употребљавају матични изрази. Ако упитна функција садржи матичне изразе, распознавање њеног типа је нешто сложеније. Да би се распознавање успешно обавило неопходно је да се дода још неколико корака, међу којима значајно место има трансформисање оригиналне упитне функције у облик у коме се распознавање може успешније одвијати. Неке од ових трансформација су привременог карактера и употребљавају се искључиво у фази распознавања типова, док су друге трајног карактера и доводе оригинални упитни израз у облик у коме се може израчунавати на серверу.

Као и у случају претходног алгоритма, и овај алгоритам се начелно може применити на различите програмске језике и базе података. Иако се у оквиру алгоритма подразумева употреба упитног језика *SQL*, он се без већих модификација може прилагодити и неком другом упитном језику. У алгоритму се претпоставља да

програмски језик располаже ламбда изразима. Ако то није случај, проблем се може решити дефинисањем по једне нове генерички именоване функције уместо сваког ламбда израза.

Делови алгоритма који су специфични за конкретну имплементацију обухватају:

- препознавање случаја да се у упиту употребљавају неки од специфичних сложених типова програмског језика;
- пресликавање препознатих сложених типова у типове програмског језика и
- имплементацију извршавања матичних израза на страни СУБП-а.

1. Препознају се имена у матичном изразу која се односе на аргументе упитне функције (тј. матичне променљиве);
2. Препознају се имена у матичном изразу која би могла се односе на атрибуте базе података који се издвајају самим упитом на *SQL*-у;
3. Помоћу СУБП-а се покуша распознавање имена атрибута и њихових типова:
   3.1 Ако се у упиту непосредно употребљавају неке матичне променљиве, онда:
      3.1.1 Свако појављивање матичне променљиве у тексту упита се замењује јединственом ознаком параметра, при чему се води таблица пресликавања матичних параметара у одговарајуће параметре;
   3.2 Упит на *SQL*-у се привремено трансформише тако да се сва имена, која би могла да се односе на атрибуте базе података, експлицитно додају у клаузулу SELECT, као нови атрибути који се читају упитом[32];
   3.3 Сваки матични израз у упиту се привремено замењује по једном новом матичном променљивом, тј. одговарајућим матичним параметром;
   3.4 Посредством управљачког модула базе података се припрема упит и распознају типови од стране СУБП-а;
   3.5 Ако је дошло до грешке у припреми упита, тада се поступак прекида и извештава се о неисправностима;
   3.6 Ако међу типовима које је СУБП препознао постоји неки од специфичних типова програмског језика, тада се сваки такав тип замењује одговарајућим типом програмског језика;
   3.7 Ако се разликују типови матичних параметара који се односе на једну исту матичну променљиву:
      3.7.1 Покушава се њихова унификација;
      3.7.2 Ако унификација не успе, значи да постоје неисправности у упиту. Поступак се прекида и извештава се о неисправностима;
   3.8 Установљавају се типови прочитаних атрибута и то како оних који се читају у оригиналном упиту, тако и оних који су додати у кораку 3.1.1;
4. Сваки матични израз који употребљава аргументе упитне функције и/или атрибуте базе података се трансформише у канонски облик примене ламбда функције на одговарајуће аргументе и атрибуте;

(наставак на следећој страни)

[32] За сва имена, која се не могу препознати као имена програмског језика у датом контексту, се претпоставља да се односе на атрибуте базе података. Ако претпоставка није тачна, то значи да се у изразу употребљава недефинисано име и у каснијим корацима (3.5) се препознају такви случајеви и извештава се о неисправностима. Детаљи имплементације са примером канонизовања матичног израза су у одељку 11.4, на страни 136.

5. Сваком матичном изразу се проверава тип:
   5.1 Проверава се тип ламбда функције која одговара изразу;
   5.2 Проверава се усаглашеност типа ламбда функције са:
      5.2.1 типовима аргумената упитне функције (који могу бити већ установљени у кораку 3, ако се непосредно употребљавају као матичне променљиве);
      5.2.2 типовима употребљаваних атрибута базе података који се преносе тој функцији (који се добијају кораком 3);
      5.2.3 очекиваним типом резултата матичног израза (који се добија кораком 3, у облику типа матичне променљиве којом је израз привремено замењен);
6. Проверава се међусобна усаглашеност типова матичних израза:
   6.1 У дефиницију упитне функције се привремено додају поддефиниције које одговарају матичним изразима:
      6.1.1 свака поддефиниција представља делимичну примену одговарајуће ламбда функције на аргументе упитне функције;
   6.2 Проверава се усаглашеност типова упитне функције и одговарајућих поддефиниција;
7. У упиту на *SQL*-у се сваки матични израз замењује одговарајућим изразом за извршавање програма на пограмском језику, при чему се програм наводи у облику ламбда функције примењене на одговарајуће матичне променљиве и атрибуте табела базе података;
8. По потреби се обезбеђују експлицитне конверзије типова аргумената и резултата позива кориснички дефинисане функције;
9. Тако трансформисан упит још једанпут пролази кроз проверу типова од стране СУБП-а ради препознавања одређених спорних случајева, које би при израчунавању могле да доведу до грешака чак и када је провера по елементима упита прошла успешно[33];
10. Ако су сви претходни кораци успешно окончани, тада је провера типова успела и типови су међусобно усаглашени. У супротном, ако у било ком кораку дође до грешке, поступак се прекида и извештава се о неисправностима.

Алгоритам 6: Проверавање типа и превођење упитних функција у којима се употребљавају матични изрази

[33] На пример, синтакса упита на *SQL*-у (*DB2 9.5*) поставља ограничење да на неким местима сме да стоји матична променљива или литерал, али не сме да стоји позив функције. У таквим случајевима упит трансформисан на описан начин не би могао да се успешно припреми ни изврши, иако је сваки део упита начелно исправан. Један пример је предикат `IN`, који као други аргумент очекује листу литерала, у којој поред литерала може евентуално да се нађе и матична променљива, али не и позив функције, па због тога ни матични израз.

Део III

Имплементација

10. Повезивање програмског језика *Wafl* и база података

10.1. Основне претпоставке

При обликовању синтаксе и семантике повезивања програмског језика *Wafl* са базама података вођено је рачуна о закључцима претходно извршених анализа, који су наведени у виду концепата у поглављу 8. Програмски језик *Wafl* обухвата интерфејс према базама података, који је независан од СУБП-а. Штавише, почива на апстрактним концептима па је у великој мери независан и од дефиниције и имплементације програмског језика.

Програмски језик *Wafl* је послужио као платформа за ово истраживање због тога што је било најпрактичније да се експериментише са језиком који задовољава основне претпоставке (строго типизиран функционални програмски језик), а чија је имплементација у потпуности доступна и погодна за експериментисање. За пратећи СУБП је одабран *IBM DB2*, из сличног разлога, јер се његова архитектура показала као довољно отворена да поднесе све неопходне експерименте и имплементацију овде представљеног решења.

Интерфејс за повезивање програмског језика *Wafl* и база података је обликован уз претпоставку да базе података представљају основни механизам за рад са трајним подацима. Почива на употреби упитног језика базе података, уз подразумевану употребу *SQL*-а, али и могућност употребе неког другог упитног језика. Постављањем упита и извођењем трансакција управља програмер, уз коришћење одговарајућег језика СУБП-а, а помоћу одговарајућих елемената програмског језика *Wafl*. Синтакса је обликована по узору на стандардизован упњеждени *SQL*.

Предност у обликовању повезивања је дата статичком начину рада са упитима и наредбама језика базе података, али је подржан и динамички начин рада. То је последица потребе да се омогући статичко типизирање програма, што је само делимично могуће у случају динамичког начина рада. У складу са тиме, у наредним одељцима се свуда, осим када је другачије експлицитно наглашено, подразумева се да је реч о статичком *SQL*-у.

Већ у фази превођења програма на *Wafl*-у може бити неопходно остваривање комуникације са базом података ради проверавања исправности типова, па чак и ради

читања делова програмског кода. Због тога се успостављањем комуникације са СУБП-ом не управља из самог програмског кода, већ се она одређује дефинисањем одговарајућих параметара од стране администратора апликације или корисника који покреће програм. Из истог разлога, један програм се може повезати само са једном базом података. Евентуално повезивање са више база података може се остварити уз примену одговарајућих могућности СУБП-а, као што су, на пример, федеративне базе података. Додатак *Конфигурисање повезивања Wafl програма са базом података*, садржи опис начина повезивања и преглед одговарајућих параметара.

У наредним одељцима су представљени концепти повезивања програмског језика *Wafl* и база података, као и опис синтаксе и начина употребе. Сви аспекти повезивања су остварени стриктно у складу са концептима представљеним у одељку 8.3. Имплементација, алгоритам проверавања типова и дискусија следе у наредним поглављима.

10.2. Упити

Основно средство за имплементацију статичког читања података из базе података у програмском језику *Wafl* су упитне функције. Упитне функције програмског језика *Wafl* представљају посебну врсту функција. Израчунавање упитне функције се састоји од израчунавања резултата упита над базом података и превођења тако добијеног резултата у облик који одговара контексту програмског језика *Wafl*.

Основни начин рада са упитима је строго типизирано лењо читање, али је подржано и слабо типизирано , као и вредно читање.

Синтакса

Синтакса дефиниције упитне функције се разликује од дефиниција других функција по томе што се као тело дефиниције наводи упитни израз:

```
<дефиниција_упита> ::=
    <име> [(<формални_параметри>)] = <упитни_израз>
    [<поддефиниције>] ;

<поддефиниције> ::=
    where { <дефиниција> {<дефиниција>} }
```

Упитни израз у програмском језику *Wafl* има синтаксу:

```
<упитни_израз> ::=
    [typed | untyped] [sql] query
    [ isolation ( s | rr | rc | ru ) ]
    { <упит_на_упитном_језику> }
```

где је `<упит_на_упитном_језику>` запис израза на упитном језику система за управљање базама података, који само чита садржај базе података.

Опциона кључна реч `sql` указује да се упит записује на упитном језику *SQL*. Концепт програмског језика *Wafl* допушта и употребу неког другог упитног језика. Конкретне имплементације могу увести подршку за друге упитне језике, при чему морају увести и нове кључне речи којима се ти језици идентификују. Ако се изостави опциона кључна реч `sql`, подразумева се да је упитни језик *SQL*. У даљем тексту се подразумева употреба упитног језика *SQL* при постављању упита.

Могућности и перформансе постављања упита у програмском језику *Wafl* су одређени употребљеним системом за управљање базама података. Уколико се конкретан систем одликује флексибилнијом и напреднијом имплементацијом упитног језика, тада и постављање упита у програмима пружа више могућности програмерима.

Тип упитне функције

Програмски језик *Wafl* подржава две врсте упита: строго типизиране и слабо типизиране. Због тога што нису подржани дословно нетипизирани упити, у дефиницијама синтаксе програмског језика се уместо пуних назива употребљавају, редом, термини *типизирани* и *нетипизирани* упити.

Типизирани упити подразумевају строгу статичку проверу сагласности типова аргумената и резултата. Нетипизирани упити омогућавају генерички рад са подацима из базе података, који се одвија независно од конкретних типова података. Врста упита се одређује опционом ознаком `typed` или `untyped`. Ако није наведена ознака врсте упита, подразумева се да је упит типизиран.

У складу са закључцима из претходних поглавља, резултат израчунавања упита представља листу редова. Начин представљања појединачних редова зависи од типизираности упита. У случају типизираних упита, сваки ред се представља одговарајућим слоговним типом, при чему сваком атрибуту резултата упита одговара један атрибут слога, са одговарајућим именом и типом. Тип резултата упита има облик:

```
List[Record[<име1>:<тип1>,...,<имеN>:<типN>]]
```

При томе се тип слога, који одговара редовима резултата упита, установљава имплицитно. Није потребно експлицитно дефинисати нове типове података да би се могли постављати одговарајући упити. У том смислу сам концепт постављања упита има у одређеној мери генеричку природу, јер се тип резултата дефинише имплицитно у складу са конкретним упитом. По томе је концепт рада са упитима у програмском језику *Wafl* сличан концепту који је примењен у систему *Kleisli*.

У случају нетипизираних упита, сваки ред се представља каталогом атрибута чији су индекси и вредности типа `String`. Атрибутима једног реда упита приступа се на начин уобичајен за каталоге. Тип резултата нетипизираног упита не зависи од атрибута који се читају и увек је:

```
List[Map[String][String]]
```

Ако упитна функција има аргументе, формална имена аргумената се могу употребљавати у оквиру записа упита у складу са правилима упитног језика. У случају *SQL*-а формална имена аргумената се могу појављивати у оквиру упита по правилима уњежденог *SQL*-а, као матичне променљиве – непосредно пре формалног имена аргумента мора се навести симбол „:“. Типови аргумената упитних функција се установљавају аутоматски.

Независно од типизираности упита, тип упитне функције се установљава аутоматски, у фази превођења. У случају типизиране упитне функције, установљава се и проверава и тип сваког прочитаног атрибута, што захтева да у фази превођења преводилац мора бити у могућности да се повеже са базом података, како би се могло остварити распознавање типова у сарадњи са СУБП-ом.

У случају нетипизираних упита се не проверавају типови аргумената. Основни разлози за то су:

- приближавање семантике нетипизираних упита динамичким упитима, чиме се омогућава једнобразно коришћење резултата упита, без обзира на број и тип прочитаних атрибута, као и
- омогућавање превођења нетепизираних упита без успостављања повезивања са базом података.

При генеричкој обради података може да се употребљава унарна функција `keys`, која израчунава листу кључева за које су дефинисане вредности у датом каталогу.

```
studentiUpisani( godina ) = typed sql query {
    select indeks, ime, prezime
    from wafldb.student
    where indeks between :godina*10000 and :godina*10000+9999
};
```

Пример 9: Строго типизирана упитна функција `studentiUpisani` издваја податке о студентима који су уписали факултет дате године[34]

Ниво изолованости упита

Ниво изолованости упита се наводи опционо, клаузулом `isolation` <*ниво*>, и може бити `s` (енгл. *Serializable*), `rr` (енгл. *Repeatable Read*), `rc` (енгл. *Read Commited*) или `ru` (енгл. *Read Uncommited*). Уколико се ниво изолованости не наведе експлицитно, подразумева се `rc` (енгл. *Read Commited*), осим ако није другачије дефинисано параметрима за подешавање апликације.

Ниво изолованости упита се одређује у складу са општим концептима повезивања (Концепт 4).

Вредно и лењо читање редова резултата упита

Упитне функције у *Wafl*-у су дефинисане са лењом семантиком. Резултат израчунавања је увек „лења листа редова“. Подразумевана употреба резултата упита се одвија уз одложено читање редова.

Да би се читање резултата упита могло одвијати и са вредном семантиком, а у складу са закључцима претходних поглавља, у програмском језику *Wafl* је омогућено експлицитно читање свих редова резултата упита, без одлагања. За то служи унарна функција `forced`. Резултат функције `forced` је једнак њеном једином аргументу, али је извесно вредно израчунат до краја, тј. у случају резултата упита не представља лењу листу. Тип функције `forced` је: `'a -> 'a`. Поред вредног израчунавања редова упита, последица примене функције `forced` може бити и затварање курсора који одговара упиту и ослобађање свих придружених ресурса који више нису потребни.

[34] Опис релација које се употребљавају у примерима се налази у додатку *Табеле које се употребљавају у примерима*, на страни 183.

10.3. Трансакције

У програмском језику *Wafl* се трансакције имплементирају у складу са одговарајућим концептима повезивања програмских језика и база података. Трансакциони механизми се односе на податке у бази података и податке на нивоу апликативне сесије[35]. Семантика трансакција је дефинисана имплицитно, конструкцијама програмског језика. О остваривању трансакционих механизама се стара имплементација програмског језика.

У складу са представљеним концептима, рад са трансакцијама у програмском језику *Wafl* се остварује кроз акционе и трансакционе функције. Трансакциона функција (трансакција) је специјализација акционе функције чије израчунавање обухвата и трансакциону логику. Трансакција може производити бочни ефекат у односу на податке у бази података и апликативној сесији. Остварен бочни ефекат се може приметити од стране других програма тек након потврђивања трансакције, што се дешава само у случају успешног израчунавања свих израза који чине трансакцију. Трансакције се могу употребљавати као самосталне функције, али и у оквиру других трансакција. Ако се нека трансакциона функција употребљава (било непосредно или посредно) у оквиру израчунавања неке друге (шире) трансакционе функције, тада се она понаша као акциона функција и њена сопствена трансакциона логика се не примењује.

Акциони изрази

Акционе и трансакционе функције се граде од акционих израза. Акциони изрази увек имају логички тип. Као акциони изрази се могу навести уобичајени функционални изрази логичког типа, али и неки од специфичних израза који се смеју употребљавати искључиво у оквиру акција и трансакција:

- наредбе упитног језика или
- наредбе `set` или `reset`.

Формална синтакса акционог израза је:

```
<акциони_израз> ::=
      | <логички_израз>
      | <наредба_упитног_језика>
      | <наредба_set>
      | <наредба_reset>

<наредба_set> ::=
      set <име> = <израз>

<наредба_reset> ::=
      reset <име>
```

Акциони израз може бити (скоро) било који исправно записан израз на програмском језику *Wafl* чија је вредност логичког типа. Тај израз може (али не мора) да непосредно или посредно обухвати израчунавање неке акционе или трансакционе функције. Једина разлика у односу на логичке изразе који нису акциони изрази јесте да акциони изрази не смеју почињати резервисаним кључним речима `set` и `reset`, као ни

[35] У основном облику, *Wafl* је прилагођен развоју апликација за Веб. У том контексту, подаци који чине стање Веб-сесије одговарају концепту података апликативне сесије и обухваћени су трансакцијама.

кључним речима којима могу почињати наредбе упитног језика. То ограничење се може једноставно превазићи употребом заграда[36]. Семантика логичких акционих израза одговара семантици одговарајућих логичких израза.

Акциони израз може бити било која наредба упитног језика, укључујући и упите. У случају *SQL*-а то може бити било која наредба *SQL*-а која се може извршити над базом података[37]. Семантика акционог израза се у том случају одређује на следећи начин:

- ако наредба представља упит, њена логичка вредност је `true` ако резултат упита има бар један ред, а `false` ако је резултат упита празан;
- ако наредба није упит, онда је њена логичка вредност `true` у случају успешног извршавања, а `false` у случају неуспешног извршавања (грешке) и
- у оба случаја, ако при извршавању наредбе дође до грешке, резултат израчунавања је `false`, а посебан регистар за грешке добија одговарајућу вредност.

Акциони израз може бити и једна од две акционе наредбе програмског језика *Wafl*. Наредба `set` израчунава дати израз <*израз*> и израчунату вредност додељује параметру (променљивој) сесије са именом <*име*>. Израз мора бити типа `String` или неког другог простог типа. У случају да тип израза није `String`, вредност израза ће се аутоматски конвертовати у тип `String` функцијом `asString` и доделити параметру сесије. То је уједно и једини случај да се у пограмском језику *Wafl* допуштају имплицитне конверзије типова.

Наредба `reset` служи за уклањање непотребног параметра сесије. Она из скупа параметара (променљивих) сесије брише параметар са називом <*назив*>.

Наредбе `set` и `reset` мењају вредности параметара апликативне сесије, али не остварују непосредан утицај на базу података. Обе наредбе су обухваћене трансакционим механизмом тако да се све до потврђивања трансакције промене не виде од стране других програма који раде у истој апликативној сесији[38]. Ове наредбе увек израчунавају вредност `true`.

Акционе функције

Акциона функција се дефинише као низ акционих израза:

```
<дефиниција_акције> ::=
    <име> [(<формални_параметри>)] =
        [typed | untyped] [sql] action
        { {<акциони_израз>;} }
        [<поддефиниције>] ;
```

[36] На пример, није допуштен акциони израз облика `update>5`, али ако се употребе заграде, добија се исправан израз: `(update>5)`.

[37] Од имплементације зависи да ли ће се омогућити само употреба наредби *SQL*-а за управљање подацима (*DML*) или и наредби за дефинисање и контролу података (*DDL* и *DCL*).

[38] Трансакциони механизми обезбеђују изолованост промена све до потврђивања трансакције, а у односу на друге програме који раде у истој апликативној сесији. Програми који раде у различитим апликативним сесијама ни под којим условима не могу делити вредности параметара сесије.

Резултат акционих функција је логичког типа. Семантика акционе функције се дефинише као конјункција низа одговарајућих логичких израза. Ова конјункција се израчунава лењо, тј. све док се израчунавањем неког израза не добије вредност `false` или се не израчунају сви изрази, управо оним редом којим су наведени у дефиницији акционе функције.

Акционе функције могу бити типизиране и нетипизиране. Врста функције се одређује опционом ознаком `typed` или `untyped`. Ако није наведена ознака, подразумева се да је функција типизирана. У типизираним акционим функцијама су све наредбе упитног језика строго типизиране. Као и у случају упитних функција, опциона кључна реч `sql` указује да се упит записује на упитном језику *SQL*.

Као и свакој другој дефиницији функције, и дефиницијама акција одговарају по два простора имена:

- простор имена дефиниције акционе функције, коме припадају формални параметри дефиниције и
- простор имена поддефиниција, коме припадају сва имена дефинисана непосредно у оквиру опционог одељка поддефиниција.

У изразима који представљају наредбе језика *SQL*, могу се као матичне променљиве појављивати само имена која су формални аргументи акционе функције, тј. имена из простора имена дефиниције. Имена из простора имена поддефиниција се могу користити у *Wafl* изразима који чине трансакциону или акциону функцију и изразима који су саставни део израза `set`.

Трансакционе функције

Трансакциона функција се дефинише као низ акционих израза који је обухваћен трансакционом логиком. Формална синтакса трансакционе функције је веома слична синтакси акционе функције:

```
<дефиниција_трансакције> ::=
    <име> [(<формални_параметри>)] =
        [typed | untyped] [sql] transaction
        [ isolation ( s | rr | rc | ru ) ]
        { {<акциони_израз>;} }
        [<поддефиниције>] ;
```

Једина разлика у односу на синтаксу акционе функције је опционо навођење нивоа изолованости трансакције. Уколико се ниво изолованости не наведе експлицитно, подразумева се ниво изолованости трансакција који је одређен за апликацију, а ако ни он није експлицитно одређен, онда се подразумева ниво изолованости *Repeatable Read*.

Као и акционе функције, и трансакционе функције имају резултат логичког типа. Вредност функције се дефинише на исти начин, као лења конјункција секвенце одговарајућих логичких израза. Међутим, семантика је сложенија, због тога што израчунавање трансакционе функције обухвата и трансакциону логику. Трансакциона логика је одређена на следећи начин:

- ако се трансакциона функција израчунава у оквиру неке друге већ започете трансакције, онда је њена семантика идентична семантици одговарајуће акци-

оне функције (тј. акционе функције са истим телом дефиниције), тј. не употребљавају се одговарајући трансакциони механизми;

- ако се трансакциона функција не израчунава у оквиру неке друге већ започете трансакције, онда:
  - започиње се нова трансакција са одговарајућим нивоом изолованости;
  - затим се израчунава привремена вредност трансакционе функције на потпуно исти начин као вредност акционе функције;
  - ако је привремена вредност трансакције `false`, трансакција се у целости поништава и коначна вредност трансакционе функције је `false`;
  - ако је привремена вредност трансакције `true`, покушава се потврђивање трансакције:
    - ако потврђивање трансакције успе, коначна вредност трансакционе функције је `true`;
    - ако потврђивање трансакције не успе, трансакција се поништава и коначна вредност функције је `false`.
- ако у било којој фази израчунавања трансакције дође до грешке, резултат израчунавања је `false`, трансакција се поништава, а посебан регистар за грешке добија одговарајућу вредност.

```
tZameniUpisanePredmete( 20070103, 2008,
                        ['M102'],
                        ['M202','R102'] )
where {
    tZameniUpisanePredmete( indeks, skgod, ispisati, upisati ) = transaction {
        ispisati
            ->map(\sifpr#indeks,skgod: aIspisiPredmet( indeks, skgod, sifpr ))
            ->aggregate( operator&&, true );
        upisati
            ->map(\sifpr#indeks,skgod: aUpisiPredmet( indeks, skgod, sifpr ))
            ->aggregate( operator&&, true );
        set poslednjiStudent = indeks;
    };

    aIspisiPredmet( indeks, skgod, sifpr ) = action {
        delete from wafldb.upisan_predmet
        where indeks = :indeks
          and sk_godina = :skgod
          and id_predmeta = (
                    select id_predmeta
                    from wafldb.predmet
                    where sifra = :sifpr
                    );
    };

    aUpisiPredmet( indeks, skgod, sifpr ) = action {
        insert into wafldb.upisan_predmet( indeks, id_predmeta, sk_godina )
        select cast(:indeks as int), id, cast(:skgod as int)
        from wafldb.predmet
        where sifra = :sifpr;
    };
}
```

Пример 10: Трансакциона функција

Претходни пример трансакционе функције ажурира податке о уписаним предметима. Најпре се бришу подаци о предметима које је потребно исписати, а затим се

уписују подаци о предметима које је потребно уписати. На крају се поставља нова вредност променљиве апликативне сесије `poslednjiStudent`.

10.4. Похрањивање података сложених типова

Иако у даљој перспективи има несумњиви значај, остваривање пуне подршке за сложене типове података уграђивањем у СУБП не представља реалну могућност у овом тренутку. Имајући у виду да је основни циљ овог истраживања сагледавање реалних могућности за повезивање релационих база података и функционалних програмских језика, подршка сложених типова у релационој бази података је остварена комбинованим приступом (одељак 7.3.3).

Сви подаци сложених типова се у бази података физички представљају у облику ниски. За записивање података се користе изрази на програмском језику *Wafl* којим се ти подаци могу записати. Изворни код се на одређен начин канонизује, како би се омогућило изостављање мање значајних садржаја и омогућило поређење једнакости међу подацима сложених типова. При томе се прављење канонизованих записа израза одвија над преведеним или израчунатим изразом, а не над оригиналним текстуалним записом израза. Тиме је омогућено да се овакав приступ примени не само на експлицитно кодиране податке већ и на било које резултате израчунавања у програмском језику *Wafl*.

Да би се при препознавању и проверавању типова, који се односе на рад са базом података, могао искористити систем типова СУБП-а, потребно је обезбедити неопходне услове за сарадњу између подсистема за проверу типова преводиоца програмског језика и одговарајућег подсистема СУБП-а. То подразумева обезбеђивање механизама за:

- препознавање сложених типова програмског језика *Wafl* као конкретних типова базе података од стране подсистема за проверавање типова СУБП-а и
- установљавање да су одређени конкретни типови базе података, које препознаје СУБП, еквивалентни одговарајућим типовима програмског језика *Wafl*.

Први услов се задовољава прављењем корисничких типова података који почивају на нискама. Ради испуњавања другог услова се дефинише пресликавање направљених корисничких типова базе података у одговарајуће типове програмског језика *Wafl*.

Имплементација похрањивања података сложених типова у бази података је описана у одељку 11.3.

Пример 9 илуструје употребу нових типова у бази података и строго типизирано читање и мењање података базе података. Претпоставља се да постоји табела `intlists`, са два атрибута[39]:

- `ID` – цео број, идентификатор листе, тип одговара *Wafl* типу `Integer`;
- `LIST` – целобројна листа, тип одговара *Wafl* типу `List[Integer]`.

[39] Пун опис табеле се налази у додатку *Табеле које се употребљавају у примерима*, на страни 183.

Програм чита из табеле листу са највећим идентификатором (функција `qPoslednjaLista`), израчунава нову листу дописивањем на почетак листе броја који је за један већи од идентификатора прочитане листе (функција `next`), па уписује такву листу у табелу (функција `write`).

При читању података се из базе података чита ниска, која представља канонски текстуални запис листе целих бројева, и аутоматски се преводи у одговарајућу интерну репрезентацију целобројне листе програмског језика. При уписивању се изводи управо обрнут поступак, јер се на основу познате листе целих бројева израчунава њена текстуална репрезентација и она се прослеђује бази података.

Оператор <*слог*>$<*име*> се употребљава за приступање елементу слога са датим именом. Име елемента се везује у тренутку превођења програма и не може бити замењено променљивом.

У овом примеру постоје две трансакције. Прва започиње имплицитно, започињањем читања података у упитној функцији `qPoslednjaLista`, а завршава се приликом завршавања употребе резултата упита, при враћању резултата функције `next`. Друга трансакција је експлицитна и обухвата израчунавање трансакционе функције `writeDb`.

```
// еквивалентно са: write(next(qPoslednjaLista()))
qPoslednjaLista()
    ->next()
    ->write()
where {
    // Функција чита листу са највећим редним бројем.
    //  -> List[ Record[ ID:Integer, LIST:List[Integer] ] ]
    qPoslednjaLista() = typed sql query {
        select *
        from intlists
        where id = (select max(id) from intlists)
    };

    // Функција next(q) на основу прочитаног реда са листом
    // израчунава листу коју је потребно додати у наредном кораку
    //  List[ Record[ ID:Integer, LIST:List[Integer] ] ] -> List[Integer]
    next(q) =
        if q->empty() then [0]
        else (q->hd()$ID + 1) : q->hd()$LIST;

    // Функције write и writeDB уписују дату листу
    // у базу података.
    //  List[Integer] -> Bool
    write(l) =
        writeDb( l->hd(), l );

    //  Integer * List[Integer] -> Bool
    writeDb( n, lst ) = typed sql transaction {
        insert into intlists
        values (:n, :lst);
        };
};
```

Пример 11: Похрањивање података сложених типова у бази података

10.5. Матични изрази

При дефинисању синтаксе матичних израза стриктно је праћен како дефинисан концепт матичних израза (Концепт 2) тако и постојећа решења употребе елемената матичног језика у упитном језику. Због тога је употреба сегмената кода програмског

језика *Wafl* у оквиру наредби *SQL*-а обликована слично употреби матичних променљивих у угњежденом *SQL*-у. Матични изрази се наводе уоквирени заградама, с тим да се пре симбола отворене заграде наводи симбол „:“:

```
<матични израз> ::= :( <израз на матичном језику> )
```

У матичним изразима се могу употребљавати различите врсте имена:

- имена из модула на програмском језику *Wafl* у коме се наводи упитни израз који садржи дати матични израз, и то:
  - име аргумента упитне (или трансакционе) функције;
  - име дефинисано у оквиру одељка поддефиниција упитне функције и
  - име дефинисано ван саме упитне функције, а видљиво према правилима видљивости програмског језика *Wafl*, као и
- имена атрибута из упитног језика, која би се могла употребити на месту на коме стоји матични израз, и то:
  - квалификована имена атрибута и
  - неквалификована имена атрибута.

Свако употребљено име се препознаје у изразу као да припада првој од наведених категорија у којој се може препознати, и то редом како су овде наведене.

Наредни пример илуструје примену матичних израза у клаузулама `SELECT` и `WHERE` упита на *SQL*-у. Издвајају се подаци о положеним предметима за студенте који су више од 10 пута излазили на испите. Претпоставља се да атрибут `ispiti` представља листу слогова, који имају атрибуте који описују појединачна полагања испита, између осталог и целобројни атрибут `ocena`:

```
select indeks, :( ispiti->filter(\r:r$OCENA>5) ) as polozeno
from ...
where :( ispiti->length() > 10 )
```

Пример 12: Матични изрази

10.6. База података као библиотека кода

При описивању начина на који се у програмском језику *Wafl* ради са подацима сложених типова, инсистирано је на општости подржаних типова. На тај начин су омогућавањем похрањивања података произвољних сложених типова у бази података створене претпоставке да се у бази података чувају и програми или потпрограми. Непосредна последица је да се база података може употребљавати као вид библиотеке програмског кода, што је у складу са усвојеним концептима (Концепт 5).

Међутим, постојећи строго типизиран концепт рада са базом података представља и одређен проблем, јер уопштени начин чувања програмског кода у табелама базе података уводи и ограничења по питању типова програма. Сасвим је јасно да се у оквиру једног атрибута једне табеле могу записивати само изрази који имају исти тип.

Програмски језик *Wafl* је проширен конструкцијама које омогућавају коришћење кода похрањеног у бази података на исти начин као што се употребљавају датотечне

библиотеке функција. Могуће је дефинисање имена различитих типова у истој библиотеци.

Употреба библиоте кода

Декларација употребе библиотеке похрањене у бази података има облик:

```
<име-библиотеке> = library db '<име-библиотеке-у-бази>';
```

При томе <*име-библиотеке*>, као и у случају употребе обичних датотечних библиотека, одређује име под којим ће се библиотека употребљавати у програму. Једина синтаксна разлика, у односу на употребу датотечних библиотека, јесте навођење кључне речи `db` уместо кључне речи `file`. Конкретно <*име-библиотеке-у-бази*> идентификује библиотеку у оквиру базе података.

Дефинисање библиотеке кода

Да би преводилац могао да употребљава библиотеку кода сачувану у бази података, она мора бити обликована стриктно у складу са спецификацијом. Спецификација библиотека је обликована тако да буде омогућено записивање сегмената кода различитих типова.

Једна библиотека се дефинише као једна релација у схеми `Wafl_lib`, где је име библиотеке идентично имену релације (без имена схеме). Структура релације која представља библиотеку обухвата два обавезна атрибута: `name` и `definition`:

- Атрибут `name` садржи име елемента библиотеке. Има тип `varchar(200) not null` (или еквивалентан). Представља примарни кључ релације.
- Атрибут `definition` садржи текст дефиниције елемента библиотеке. Има тип `long varchar not null` (или неки сличан дугачак тип ниски).

Сваки ред библиотеке представља дефиницију једног елемента библиотеке, облика:

```
<име> = <дефиниција>
```

При томе:

- атрибут `name` садржи непразну ниску, која представља <*име*>;
- атрибут `definition` садржи тело дефиниције <*дефиниција*> (без симбола „;“ на крају) и
- елементи библиотеке се дефинишу искључиво у облику именованих израза, без експлицитног навођења аргумената, па се функције морају дефинисати искључиво применом ламбда израза;
- због вишекорисничке природе базе података, у случају програмског језика *Wafl* је уведено ограничење да дефиниције у библиотеци кода у бази података морају бити међусобно независне, тј. да се не могу међусобно употребљавати[40].

[40] Иако ово може да изгледа ограничавајуће, у пракси се не осећа због тога што канонизован запис функције, који се преноси бази података при пуњењу библиотеке из програмског језика, обухвата и све потребне поддефиниције. То има за последицу одређено повећавање записаног кода, али и међусобну независност елемената библиотеке.

Свака библиотека може садржати и највише један ред у коме је вредност атрибута `name` празна ниска. Такав ред представља документациони ред библиотеке. Атрибут `definition` садржи произвољан текст са информацијама о библиотеци.

Поред наведених обавезних атрибута дозвољено је додавање додатних атрибута ради записивања детаљнијих информација о елементима библиотеке. Препоручује се дефинисање следећих опционих атрибута:

- `description`, неког текстуалног типа дужине бар 200 знакова, садржи опциони опис елемента библиотеке;
- `author`, неког текстуалног типа дужине бар 200 знакова, садржи опционе информације о аутору елемента библиотеке и
- `version`, неког текстуалног типа дужине бар 20 знакова, садржи опционе информације о верзији кода елемента библиотеке.

Пример библиотеке кода у бази података

Пример 13 илуструје прављење библиотеке кода у бази података. Библиотека садржи неколико једноставних дефиниција константи и функција. У овом једноставном примеру библиотека се прави коришћењем скрипта на *SQL*-у. најпре се прави табела `wafl_lib.testdblib` и додељују се права коришћења свим корисницима базе података. Затим се експлицитно додају дефиниције. Пример 14 садржи еквивалентну библиотеку дефинисану помоћу датотеке. Пример 15 илуструје начин употребе претходно дефинисане библиотеке.

```
create table wafl_lib.testdblib(
    name        varchar(200) not null,
    definition  long varchar not null,
    description long varchar,
    primary key(name)
);

grant select
    on table wafl_lib.testdblib
    to public;

insert into wafl_lib.testdblib( name, definition )
values
    ( 'intvalue', '25' ),
    ( 'verzija', '''0.1''' ),
    ( 'id', '\x:x' ),
    ( 'idx', 'id where {id(x)=x;}' ),
    ( 'add', '\x,y:x+y' );
```

Пример 13: Прављење библиотеке кода у бази података

```
library testdblib {
    intvalue = 25;
    verzija = '0.1';
    id = \x:x;
    idx = id where {id(x)=x;};
    add = \x,y:x+y;
}
```

Пример 14: Датотечна библиотека еквивалентна претходној библиотеци дефинисаној у бази података

```
{#
    dblib::intvalue,
    dblib::verzija,
    dblib::id('aaa'),
    dblib::idx(1),
    dblib::add(7,3)
#}

where {
    dblib = library db 'testdblib';
}
```

Пример 15: Употреба библиотеке кода у бази података

10.7. *Динамички SQL*

Концепти примене динамичког *SQL*-а у *Wafl*-у се не разликују значајно од уопштеног концепта динамичког *SQL*-а, који је установљен стандардом упитног језика *SQL*.

Текст *SQL* наредби које се динамички израчунавају наводи се у облику ниске, а евентуални параметри у облику каталога параметара чије су вредности такође ниске. Динамички упити морају бити нетипизирани јер у тренутку превођења програма није познат текст упита, а тиме ни тип његовог резултата. Начин употребе резултата упита је исти као у случају статичких нетипизираних упита. Тип резултата динамички постављеног упита се употребљава на сличан начин као резултат статичког нетипизираног упита.

Функција `dynSqlQuery(q)` омогућава основни начин извршавања динамичких упита. Има један аргумент – ниску која представља упит. Резултат је структура која садржи податке о успешности извршавања упита и евентуалним грешкама, као и нетипизирани резултат извршавања упита. Тип функције `dynSqlQuery` је:

```
String -> Record[
            ok:Bool,
            errCode:Integer,
            errText:String,
            result:List[Map[String][String]]
            ]
```

Функција `dynSqlQueryP(q,p)` омогућава употребу параметара упита. Тип ове функције је:

```
String * Map[String][String]
    -> Record[
            ok:Bool,
            errCode:Integer,
            errText:String,
            result:List[Map[String][String]]
            ]
```

У тексту динамичког упита се параметри наводе на исти начин као у случају статичких упита, с тим да се сви параметри задају у виду ниски. За задавање параметара се употребљава други аргумент функције `dynSqlQueryP`. Кључеви каталога су називи параметара, а вредности каталога су вредности одговарајућих параметара.

У случају исправног извршавања упита резултат је:

```
{   ok = true;
    errCode = 0;
```

```
        errText = '';
        result = <резултат упита> }
```

У случају грешке резултат је:

```
    {   ok = false;
        errCode = <нумерички код грешке>;
        errText = <текстуални опис грешке>;
        result = <генерички резултат упита са описом грешке> }
```

Тада `<генерички резултат упита са описом грешке>` представља листу каталога са информацијама о насталим грешкама.

За динамичко извршавање наредби упитног језика се употребљавају функције `dynSqlCommand` и `dynSqlCommandP`. Функција `dynSqlCommand(c)` има тип:

```
String -> Record[
            ok:Bool,
            errCode:Integer,
            errText:String,
            ]
```

и израчунава наредбу *SQL*-а записану у оквиру ниске `c`. Наредба не сме да садржи матичне променљиве. Слично, функција `dynSqlCommandP(c,p)`, чији је тип:

```
String * Map[String][String]
    -> Record[
            ok:Bool,
            errCode:Integer,
            errText:String,
            ]
```

израчунава наредбу *SQL*-а записану у оквиру ниске `c`, с тим да наредба може да садржи матичне променљиве, чије се вредности наводе у оквиру аргумента `p`.

Обе функције су акционе и не смеју се употребљавати ван оквира неке трансакције. У случају успешног извршавања израчунавају структуру:

```
    {   ok = true;
        errCode = 0;
        errText = '' }
```

У случају грешке, резултат је:

```
    {   ok = false;
        errCode = <нумерички код грешке>;
        errText = <текстуални опис грешке> }
```

Ако би имплементација програмског језика *Wafl* подржавала и неки други упитни језик (поред *SQL*-а), било би потребно да се обезбеде и функције за динамичко извршавање наредби тог упитног језика.

Иако постоје читаве класе проблема који се могу решавати искључиво применом динамичког *SQL*-а, у овом раду је пажња скоро искључиво посвећена статичком начину повезивања програма и базе података. Разлог за то је у чињеници да се све анализе типова података у *Wafl*-у обављају статички, па се и типови израза који укључују динамичке упите морају препознати статички, док још нису познати упити који ће се динамички израчунавати. Због тога се са динамичким упитима и наредбама може радити само слабо типизирано. Самим тим, све оно што се односи на нетипизирани

динамички начин рада је већ обрађено кроз слабо типизиран статички начин повезивања са базом.

У приложеном сегменту програма се динамички *SQL* употребљава за издвајање података по услову који је одређен аргументом функције. Конкретан услов издваја податке о студентима у чијем имену постоји мало или велико слово *A*.

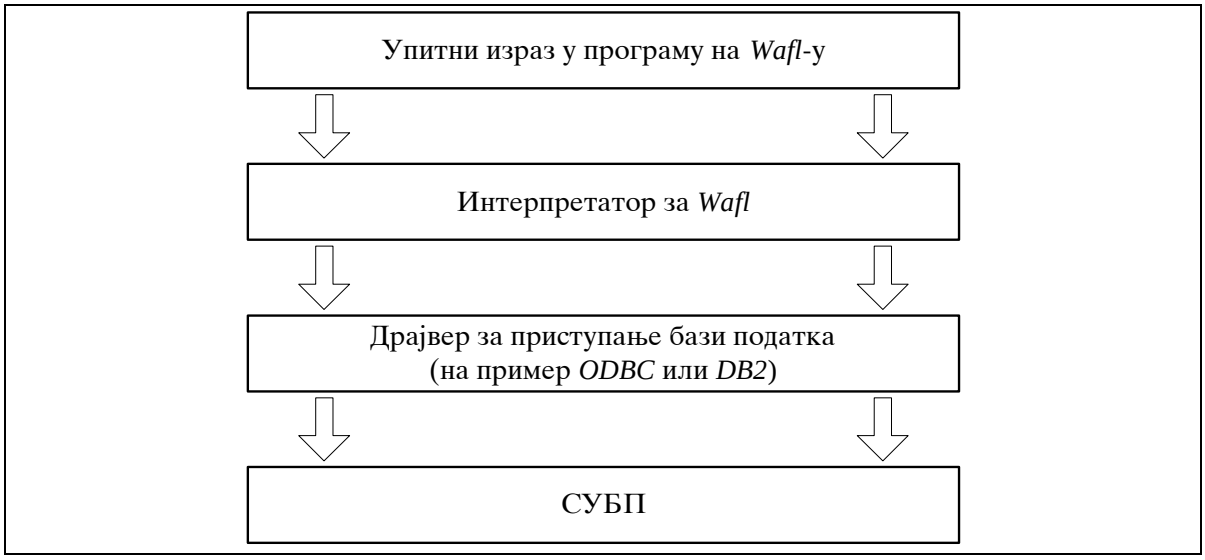
```
studentiUslov( "lower(ime) like '%a%'")
where {
    studentiUslov( uslov ) =
        dynSqlQuery(
            "select indeks, ime, prezime
            from student
            where " + uslov
        );
}
```

Пример 16: *Динамички SQL*

11. Имплементација повезивања

11.1. Архитектура повезивања са СУБП-ом

Интерфејс експерименталне имплементације програмског језика *Wafl* према базама података је пројектован модуларно, по слојевима, са циљем да се омогући употреба различитих система за управљање базама података. Сви основни елементи повезивања су апстраховани. Слика 3 приказује слојеве имплементације интерфејса.



Слика 3: Функционални дијаграм подсистема за рад са базом података

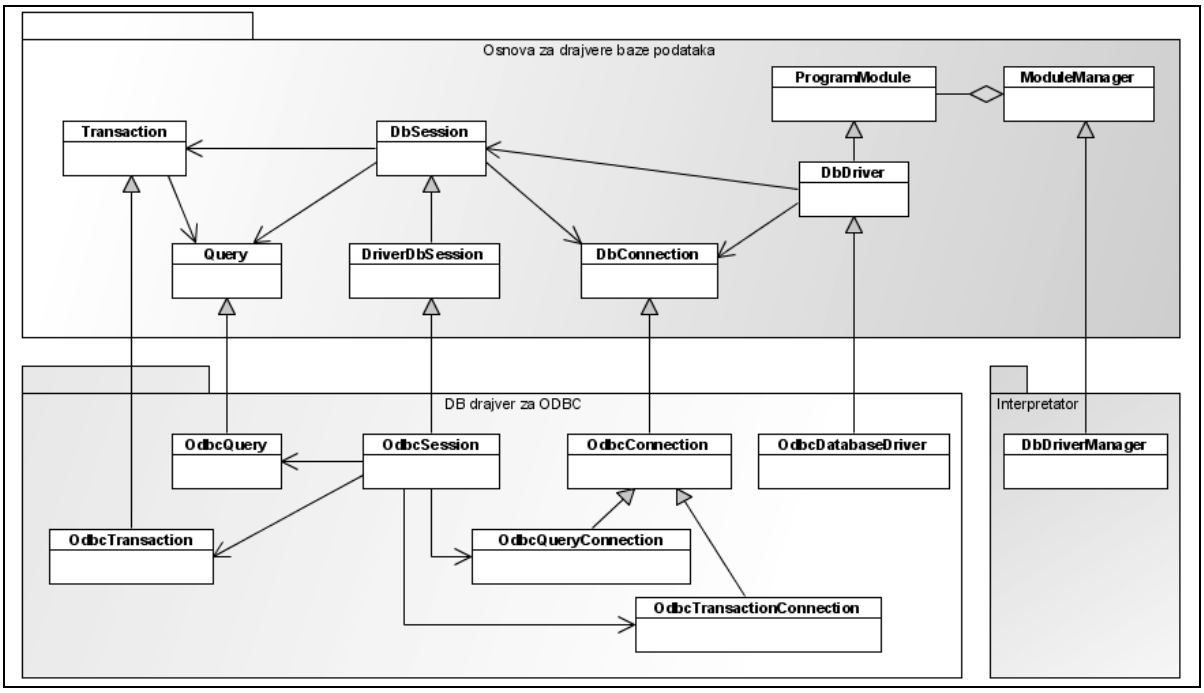
Апстрактан концепт драјвера за базу података почива на још општијем концепту модула. Инстанце апстрактног драјвера имплементирају елементе нижег слоја комуникације са базом података, који су специфични за конкретне имплементације РСУБП-а. Слика 4 представља оквирни *UML* дијаграм класа подсистема за приступање бази података који илуструје унутрашњу архитектуру имплементације.

Апстрактан драјвер обухвата и друге апстрактне концепте, као што сваки конкретан драјвер обухвата одговарајуће елементе тих концепата који су специфични за један конкретан начин повезивања са СУБП-ом. У апстраховане концепте, поред драјвера спадају и:

- *Веза са базом података*, која обезбеђује све основне елементе комуникације програма и базе података (апстрактна класа `DBConnection`);
- *Упит* на *SQL*-у, који се поставља над базом података (апстрактна класа `Query`);
- *Трансакција* на нивоу базе података (апстрактна класа `Transaction`) и
- *Сесија*, која представља једну остварену веза програма и базе података (апстрактна класа `DbSession`).

Свака сесија представља посебну инстанцу комуникације са базом података. Изолована је од других сесија у мери у којој то обезбеђује употребљени ниво изолованости. Из угла имплементације интерпретатора, сесија представља кључни објекат интерфејса према бази података, јер се путем сесије праве и употребљавају сви други објекти, попут упита и трансакција.

Управљање радом подсистема обезбеђује *управљач драјверима* (класа `DbDriverManager`), који остварује повезивање програма са базом посредством одговарајућег драјвера, а затим даље старање о комуникацији усмерава на тако направљену везу са базом. Управљач драјверима за базу података представља специјализацију уопштеног управљача модулима.



Слика 4: Дијаграм класа подсистема за рад са базом података, на примеру класа за драјвер за *ODBC*

У оквиру експерименталне имплементације је имплементирано више драјвера, међу којима су најважнији драјвери за:

- *ODBC* – применом овог драјвера се обезбеђује повезивање са било којим СУБП који подржава *ODBC* и
- *DB2* – више драјвера за различите верзије СУБП *DB2*, који омогућавају непосреднију и ефикаснију комуникацију са системом *DB2* у односу на уопштени драјвер за *ODBC*.

У оквиру управљања комуникацијом са базом података, улога драјвера је да обезбеди исправну размену информација између програма и базе података. То подразумева и трансформисање података из облика у коме их употребљава СУБП у облик у коме их употребљава интерпретатор, као и обратно. Драјвер има значајну улогу и у поступку превођења. Посао драјвера је да обезбеди, путем комуникације са СУБП-ом, неопходне информације о типовима, да би било могуће установити тип упитих израза и проверити усаглашеност типова у програмима.

Успостављање комуникације између програма писаног на програмском језику *Wafl* и СУБП-а се одвија аутоматски, на основу вредности одговарајућих конфигурационих параметара апликације. Програм може успостављати комуникацију само са једном базом података. Један исти програм може (имплицитно и према потреби) успоставити више линија комуникације (тј. *веза*) са истом базом података. При томе се тренутно неупотребљаване везе могу чувати (да се у случају поновљене потребе не би чекало на њихово поновно успостављање) и аутоматски затварати када њихов број пређе границу дефинисану одговарајућим конфигурационим параметром апликације.

За сваку трансакцију се обезбеђује посебна веза са одговарајућим нивоом изолованости. Различите трансакције истог нивоа изолованости могу делити исту везу уколико су дисјунктне у времену (тј. извршавају се строго једна после друге). Подразумевани ниво изолованости упита је *Read Commited*, али се ниво изолованости упита може променити, како на нивоу читаве апликације, тако и на нивоу појединачног упита.

11.2. Подсистем за управљање трансакцијама

Увођење концепта трансакција у сам програмски језик представља значајну специфичност програмског језика *Wafl*. Због тога је подсистем за управљање трансакцијама је један од најважнијих делова извршног окружења.

Подсистем за управљање трансакцијама, који обезбеђује јединствен трансакциони простор за базу података, датотеке и параметре сесије је технички веома сложен, пре свега због тога што системи за управљање датотекама углавном не нуде подршку за трансакције. Како је овде представљено истраживање било усмерено пре свега према базама података, имплементиран је трансакциони простор који обухвата све промене на бази података и на параметрима сесије, али не и на датотекама. Имплементација јединственог трансакционог простора је заснована на следећим претпоставкама:

- трансакције које обухватају само податке из базе података могу да се изводе без примене дистрибуираног управљања трансакцијама и двофазног потврђивања трансакција;
- потврђивање трансакција које су само над параметрима сесије практично увек успева;
- трансакције које су над базом података и параметрима сесије не морају бити засноване на дистрибуираном управљању трансакцијама, па ни на двофазном потврђивању трансакција, јер су параметри сесије у оквиру локалног подсистема на коме потврђивање трансакција практично увек успева.

Због тога је уместо примене двофазног потврђивања трансакција довољно да се најпре покуша потврђивање дела трансакције над базом података, па ако то успе, онда се она потврди и у подсистему за старање о параметрима сесије.

Имплементација подсистема за управљање трансакцијама у програмском језику *Wafl* почива на следећим претпоставкама:

- Препознавање почетка трансакције се одвија имплицитно, самим започињањем израчунавања трансакционе функције.
- Ради подизања ефикасности рада са параметрима сесије, свака трансакција прави локалну копију параметара сесије којима је претходно већ приступала.
- Извођење промена над базом података се остварује у оквиру одговарајуће трансакције СУБП-а.
- Извођење промена над параметрима сесије се одвија над локалним копијама вредности параметара сесије.
- Трансакција се потврђује аутоматски, у случају успешног израчунавања трансакционе функције.
- Трансакција се поништава аутоматски, у случају неуспеха при израчунавању трансакционе функције.
- Изоловање трансакције се постиже закључавањем употребљаваних параметара сесије и мењањем њихових локалних копија.
- Операције над параметрима сесије се обављају у атомичним блоковима, тако да у једном тренутку подацима једне сесије може непосредно приступати највише једна трансакција.

У тексту који следи су представљени алгоритми за читање и мењање параметара сесије, као и за потврђивање и поништавање трансакција.

1. Да ли у оквиру трансакције постоји локална копија траженог параметра сесије?
2. Ако не постоји:
   2.1 Закључава се сесија.
   2.2 Да ли на траженом параметару сесије постоји катанац *X* неке друге трансакције?
   2.3 Ако не постоји:
      2.3.1 На тражени параметар сесије се ставља катанац *S*.
      2.3.2 Преписује се вредност параметра сесије у локалну копију.
   2.4 Иначе, ако постоји:
      2.4.1 Откључава се сесија.
      2.4.2 Чека се на откључавање параметра сесије. У случају истека допуштеног времена чекања, пријављује се грешка и прекида израчунавање.
      2.4.3 Наставља се од корака 2.1.
   2.5 Откључава се сесија.
3. Чита се вредност локалне копије параметра сесије и његова вредност је резултат операције читања.

Алгоритам 7: Читање параметара сесије

1. Да ли у оквиру трансакције постоји локална копија траженог параметра сесије са ознаком да је мењана?
2. Ако не постоји:
   2.1 Закључава се сесија.
   2.2 Да ли на траженом параметару сесије постоји катанац неке друге трансакције?
   2.3 Ако не постоји:
      2.3.1 На тражени параметар сесије се ставља катанац *X*.
      2.3.2 Преписује се параметар сесије у локалну копију и означава да је мењан.
   2.4 Иначе, ако постоји:
      2.4.1 Откључава се сесија.
      2.4.2 Чека се на откључавање параметра сесије. У случају истека допуштеног времена чекања, пријављује се грешка и прекида израчунавање.
      2.4.3 Наставља се од корака 2.1.
   2.5 Откључава се сесија.
3. Мења се вредност локалне копије параметра сесије.

Алгоритам 8: Мењање података сесије

1. Ако је у трансакцији приступано параметрима сесије:
   1.1 Закључава се сесија.
2. Потврђује се трансакција у бази података.
3. Ако је потврђивање успело и ако су мењане локалне копија неких параметара сесије, тада се све промене локалних копија параметара сесије преписују у сесију.
4. Ако је у трансакцији приступано параметрима сесије, тада:
   4.1 Повлаче се сви катанци са параметара сесије.
   4.2 Бришу се локалне копије параметара сесије.
   4.3 Откључава се сесија.

Алгоритам 9: Потврђивања трансакција

1. Ако је у трансакцији приступано параметрима сесије:
   1.1 Закључава се сесија.
2. Поништава се трансакција у бази података.
3. Ако је у трансакцији приступано параметрима сесије:
   3.1 Бришу се локалне копије параметара сесије.
   3.2 Повлаче се сви катанци са параметара сесије.
   3.3 Откључава се сесија.

Алгоритам 10: Поништавање трансакција

11.3. Записивање података сложених типова

Физичка репрезентација података

Подаци сложених типова, који нису подржани од стране СУБП-а, се у бази података физички представљају у облику канонизованих текстуалних записа израза на програмском језику *Wafl*. Вредна природа израза у програмском језику *Wafl* има за последицу да се при преношењу података од програма према бази података сваки израз

најпре израчунава[41], па се затим аутоматски преводи у канонизован текстуални облик и тек онда прослеђује као аргумент упитне или трансакционе функције.

Канонизованост записа подразумева да ће записи садржати генеричка имена уместо оригиналних имена наведених у програму, као и да ће све константе бити записане на уједначен начин.

Све дефиниције које се употребљавају у изразу се наводе у одговарајућем блоку *where*. Имена дефинисаних функција или израза се одређују генерички, у облику:

f<*редни_број*>

при чему се <*редни_број*> одређује по редоследу појављивања у телу израза. Имена формалних аргумената функција се такође одређују генерички у облику:

a<*редни_број*>

где <*редни_број*> одговара редном броју аргумента у дефиницији.

Сложени типови у бази података

При размени података сложених типова између компоненти програмског језика и СУБП-а, подаци се преносе у облику ниски. Да би СУБП био у стању да препозна да се ради о различитим сложеним типовима, а не о генеричком типу ниски, на нивоу СУБП-а се употребљавају кориснички типова података, који почивају на нискама.

То је уједно и једини постојећи механизам база података који се у пракси може употребити у те сврхе. Евентуалну алтернативу би могло да представља дефинисање сложених структурних типова базе података. Међутим, иако би то могло бити флексибилније у смислу касније употребе података у бази података, ипак не представља довољно опште решење, јер је систем типова функционалног програмског језика много богатији него чак и тако проширен систем типова базе података. Са друге стране, специфичност имплементације структурних типова у различитим СУБП-овима би представљала проблем у размени података и захтевала различите физичке репрезентације података у зависности од начина њиховог дефинисања у бази података.

Због тога што је за физичку репрезентацију података сложеног типа одабран тип знаковних ниски, сваки кориснички дефинисан тип мора да почива на неком типу ниски. У зависности од конкретног типа и очекиване величине података може се одабрати тип ниски који набоље одговара потребама.

Систем *DB2*, као и већина других савремених система за управљање базама података, омогућава дефинисање корисничких типова података (енгл. *user defined types*) чија физичка имплементација почива на неким постојећим, већ подржаним типовима. Систем *DB2* располаже са четири основна типа ниски [IBM:2008]:

- `CHAR(`*n*`)` – знаковна ниска задате дужине *n* знакова, где је *n* од 1 до 254;
- `VARCHAR(`*n*`)` – знаковна ниска променљиве дужине, са задатом највећом дужином од *n* знакова, где је *n* до 32672;

[41] Израчунавање би у идеалном случају требало да има за резултат нормалну форму израза, али због вредног израчунавања то неће увек бити случај, јер израчунавање ниједне функције неће бити започето уколико нису обезбеђени сви аргументи. У пракси резултат израчунавања има тзв. *слабу нормалну форму* (енгл. *weak head normal form – WHNF*).

- `LONG VARCHAR` – ниска променљиве дужине од највише 32700 знакова;
- `CLOB(`*n*`)` – знаковна ниска променљиве дужине од највише *n* знакова, где је *n* до 2.147.483.647 знакова.

Кориснички типови података се праве наредбом *SQL*-а:

```
create distinct type <нови_тип>
    as <постојећи_основни_тип>
    [ with comparisons ]
```

Опциона декларација `with comparison` наглашава да је могуће поредити вредности нових типова. Обавезна је за све типове осим тзв. дугачких типова[42]. У случају дугачких типова није подржано поређење, па је опција забрањена.

Који ће од типова базе података бити одабран за неки конкретан тип програмског језика зависи од процењене очекиване горње границе величине канонизованог облика записа. У случају малих података може бити довољно да се користи неки од основних тпова, на пример `varchar(100)`, али у случају великих података може бити неопходна употреба неког од већих типова, на пример `clob(250M)`.

Повезивање сложених типова базе података и програмског језика

Везивање конкретних типова базе података, које препознаје СУБП, и еквивалентних типова програмског језика *Wafl* се имплементира дефинисањем одговарајућег пресликавања типова. Такво пресликавање се може дефинисати на више разиличитих начина. Због одабраног концепта комбинованог повезивања најбоље је да се пресликавање типова дефинише у облику табеле базе података.

Веома је значајно што се и кориснички типови података и пресликавање типова остварују у истој бази података у којој се налазе и подаци који се употребљавају у програмима, јер важење ових типова и јесте ограничено управо на ту базу података. Свако решење које би било лоцирано ван те базе података остављало би простор за појављивање типовних грешака у току рада програма.

Други квалитет таквог решења је његова отвореност, јер омогућава корисницима базе података да помоћу уобичајених алата базе података дефинишу нови кориснички тип и пресликавање тог типа у тип програмског језика.

Одговарајућа табела се дефинише на следећи начин:

```
create table wafl.dtypes (
    DistinctType    varchar(128) not null,
    WaflType        varchar(4000) not null,
    primary key ( DistinctType )
)
```

Сваки ред табеле `wafl.dtypes` одређује да датом кориснички дефинисаном типу, са именом `DistinctType`, одговара дати тип програмског језика *Wafl*, чији је запис наведен као вредност атрибута `WaflType`. Дужина атрибута `DistinctType` је усклађена са највећом допуштеном дужином имена кориснички дефинисаног типа у систему *DB2*.

[42] Дугачки типови су они чије се вредности у систему *DB2* не чувају физички на истом месту као и остали подаци. У дугачке типове спадају: `LONG VARCHAR`, `LONG VARGRAPHIC`, `BLOB`, `CLOB` и `DBLOB`.

Дужина атрибута `WaflType` је одређена као процењена разумна горња граница за дужину записа типа података, али може се и повећати.

Да би ово пресликавање било двосмерно (тј. да би било бијективно), потребно је остварити јединственост типова програмског језика. Међутим, систем *DB2* има ограничење дужине јединствених кључева, због чега то није могуће остварити помоћу јединственог кључа табеле, већ се о томе мора водити рачуна при дефинисању и евидентирању типова.

Софтверске компоненте

За успешну комуникацију интерпретатора и програма написаног на *Wafl*-у са базом података задужен је управљачки модул (драјвер) за приступање бази података. Управљачки модул је задужен да:

- у фази препознавања типова обезбеђује одговарајуће превођење типова;
- при извршавању програма преводи прочитане податке из текстуалног записа у одговарајући интерни облик програмског језика *Wafl* и
- при превођењу прочитаних података проверава да ли њихов стварни тип одговара декларисаном типу.

У фази превођења програма, управљачки модул је задужен за извршавање дела алгоритма препознавања типова који се односи на типове аргумената и резултата упитних функција.

У току извршавања програма управљачки програм мора да сваки прочитан податак испоручи програму у облику у коме га он може непосредно употребљавати. У случају неког кориснички дефинисаног типа, у овој фази је неопходно обавити обрнут поступак од онога који се одвија при превођењу података у њихову физичку репрезентацију. Текстуални записи података се преводе у одговарајуће интерне репрезентације података програмског језика. Тај поступак превођења се у основи не разликује од превођења које обавља преводилац програмског језика, због чега управљачки модули за то користе услуге одговарајућих модула преводиоца. Управљачки модул испоручује податак модулу за превођење у похрањеном (текстуалном) облику. Модул за превођење преводи запис податка и установљава његов тип, па добијени превод и податке о типу враћа управљачком модулу, који проверава да ли установљен тип одговара типу који је за тај података евентуално дефинисан у бази података.

Циљ додатне провере исправности типа у фази извршавања је да се препознају евентуалне неисправности података, до којих може доћи ако је садржај базе података мењан од стране других програма, писаних на другим програмским језицима, а да при томе није проверавана исправност записа и њихова усклађеност са декларисаним типом.

11.4. Имплементација матичних израза

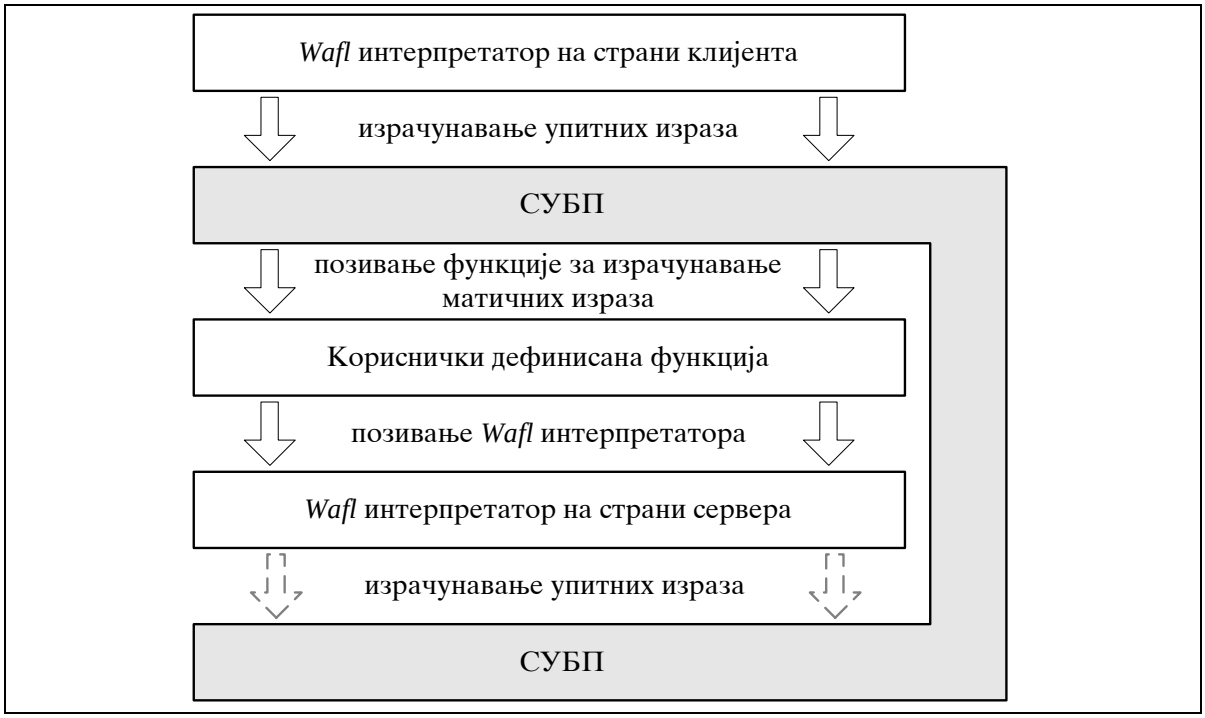
Извршавање матичних израза на самом серверу, у оквиру СУБП-а, мора се одвијати уз остваривање сарадње између компоненти преводиоца за матични језик и СУБП-а. Овај део имплементације је остварен специфично за СУБП *IBM DB2*. Имплементација се ослања искључиво на службено документоване механизме система *IBM DB2*. Због тога што већина савремених СУБП располаже одговарајућим механизмима, начелно је

могуће имплементирати одговарајући подсистем за извршавање матичних израза и на другим СУБП-овима, без икаквих концептуалних измена.

Основно средство за извршавање неког програмског кода у оквиру СУБП-а су серверске процедуре и кориснички дефинисане функције. Због тога што израчунавање матичног израза на програмском језику *Wafl* по својој природи представља израчунавање израза који има неку вредност, у овом случају је као природно решење одабрана употреба кориснички дефинисаних функција.

Израчунавање матичних израза се одвија на следећи начин:

- интерпретатор програмског језика *Wafl*, посредством подсистема за управљање повезивањем са базама података и одговарајућег управљачког модула, прослеђује СУБП-у упитни израз који је потребно извршити, а у коме се налазе матични изрази;
- СУБП позива кориснички дефинисане функције ради израчунавања матичних израза;
- кориснички дефинисане функције се понашају као међуслој за позивање интерпретатора програмског језика *Wafl* на серверу и
- ако у матичном изразу постоји неки упитни израз, интерпретатор програмског језика *Wafl* за његово израчунавање користи услуге СУБП-а, на исти начин као што то ради и интерпретатор на страни клијента.



Слика 5: Функционални дијаграм израчунавања матичних израза

Да би сваки од ових корака био остварив неопходно је најпре извршити одређене трансформације матичног израза и упитног израза у коме се матични израз употребљава.

Начин преношења матичних израза

Да би кориснички дефинисана функција могла да изврши матични израз на *Wafl*-у, неопходно је да имплементацији те функције буду доступни тај израз и одговарајући алати за извршавање *Wafl* израза.

Основни начин је да се матични израз на програмском језику *Wafl* достави кориснички дефинисаној функцији јесте да се проследи у облику текстуалног записа. Непосредна последица таквог начина рада је да се изрази на серверу морају поново преводити, па овакав начин размене информација није најефикаснији. Са друге стране, овакав приступ омогућава да се матични изрази преносе у истом облику у коме се изрази на *Wafl*-у записују у бази података, што може значајно да помогне у имплементацији.

Други начин би био да се израз преноси у неком преведеном или полупреведеном облику. У зависности од облика у коме се код преноси, може бити неопходно да се на страни сервера обаве одређена повезивања са системским и библиотечким функцијама програмског језика. На серверу се поступак превођења матичног израза свакако не понавља у целости, па овакав начин преношења матичних израза пружа боље перформансе. Ипак, у случају његове имплементације је неопходно узети у обзир да би могло доћи до одређених проблема, ако постоје неке функционалности које су допуштене на клијенту а забрањене на серверу (на пример, рад са локалним датотекама).

У оквиру обављеног истраживања је преношење израза извођено у елементарном текстуалном облику. Иако су тиме потенцијално умањене перформансе, у овој фази истраживања је далеко већи значај придаван једноставнијем и поузданијем обављању провера исправности у свим фазама трансформисања, превођења и израчунавања израза.

Канонски облик матичних израза

Поред начина преношења матичних израза, значајно је и какав облик морају да имају матични изрази да би било могуће њихово израчунавање на страни сервера. Ако се посматрање ограничи на синтаксу матичних израза, онда је јасно да би требало да се као матични израз може употребити сваки исправан израз програмског језика *Wafl*, чији резултат има тип који се може исправно протумачити од стране упита на *SQL*-у.

Међутим, семантика матичних израза намеће одређена ограничења. Проблем настаје као последица могућности да се у матичном изразу употребљавају аргументи упитне функције и вредности атрибута које се добијају упитом. Употребљена имена дефинисана у другим деловима програма на *Wafl*-у, као и изрази или вредности које одговарају тим именима, могу да се повезују статички при превођењу матичних израза, али је то далеко сложеније остварити са аргументима упитне функције, а практично немогуће са вредностима атрибута. Због тога сви матични изрази морају бити аутоматски трансформисани у облик чији се тип може статички установити.

Решење примењено у имплементацији матичних израза у програмском језику *Wafl* је да се матични израз трансформише у примену ламбда функције. Матични израз се трансформише тако да:

- телу функције одговара тело матичног израза;

- свако појављивање имена атрибута у телу функције се замењује новим генеричким именом аргумента, облика **_a**<*редни број*>;
- аргументи функције се одређују тако да:
  - сваком аргументу упитне функције, који се употребљава у матичном изразу, одговара по један аргумент функције са управо таквим именом;
  - сваком имену атрибута неке табеле из базе података, који се употребљава у матичном изразу, одговара по један аргумент функције са одговарајућим генеричким именом;
  - најпре се наводе сви аргументи који одговарају аргументима упитне функције, па затим сви аргументи који одговарају атрибутима који се читају упитом;
- примена функције се одређује као примена тако обликоване функције на одговарајуће аргументе упитне функције и атрибуте базе података.

Добијени израз представља канонски облик матичног израза.

На пример, у приложеној упитној функцији `q`:

```
q(a,b,c) = typed sql query {
    select :( a + t.attr1 + c ) as zbir
    from schema.table t
    };
```

матични израз се трансформише на следећи начин:

```
q(a,b,c) = typed sql query {
    select
        :( (\a,c,_a1: a + _a1 + c )(a,c,t.attr1) )
            as zbir
    from schema.table t
    };
```

Описаном трансформацијом је омогућено да се применом одговарајуће кориснички дефинисане функције базе података, уз непосредно навођење одговарајућих аргумената, израчуна трансформисани матични израз. Кориснички дефинисана функција мора као аргументе да добије текст трансформисаног матичног израза и сваки појединачан аргумент. Аргументи упитне функције се наводе у облику матичних променљивих, са префиксним симболом ':', док се атрибути наводе без посебног означавања.

Ако се претпостави да се та кориснички дефинисана функција зове `WaflEvaulate`, онда је коначан облик претходне упитне функције:

```
q(a,b,c) = typed sql query {
    select WaflEvaluate(
                '\a,c,_a1: a + _a1 + c',
                :a, :c, t.attr1
            ) as zbir
    from schema.table t
    };
```

Посебан квалитет представљеног раздвајања тела матичног израза од аргумената је што се на тај начин омогућава да се тело матичног израза на серверу преведе само једанпут, без обзира на то колико ће се пута и са каквим аргументима израчунавати

одговарајућа упитна функција, као и на то колико ће се пута матични израз израчунавати при једном израчунавању упита.

Алтернатива оваквом приступу би могла бити да се матични израз не трансформише, већ да се пре његовог израчунавања, у телу функције, изврши замена одговарајућих референци на аргументе упитне функције и атрибуте упита њиховим актуелним вредностима. Тада би уз сваку матичну променљиву морало да се наведе и њено име, како би се могла извршити замена. Слично томе, и вредности атрибута би морале да буду праћене именом.

Иако може деловати привлачно да се проблем реши без трансформисања оригиналног израза, то би заправо представљало само одлагање проблема. У изразу који је потребно израчунати је свакако неопходно да се у неком тренутку замене актуелне вредности употребљаваних аргумената и атрибута. Због тога би чак и уз овакав приступ било неопходно примењивати претходно представљено трансформисање матичног израза, и то не само једанпут при превођењу, већ сваки пут при израчунавању тог израза.

Кориснички дефинисане функције

У претходним одељцима је описано на који начин се кориснички дефинисане функције употребљавају за израчунавање матичних израза на програмском језику *Wafl*. Одређен проблем представља чињеница да матични изрази трансформисани у ламбда функције могу имати различит број аргумената, а кориснички дефинисане функције система *DB2* немају могућност преношења променљивог броја аргумената. Једно решење је да се имплементира функција са довољно великим бројем аргумената и затим употребљава са експлицитним навођењем неких посебних вредности неупотребљених аргумената. Други начин је да се за сваки подржан број аргумената направи посебна кориснички дефинисана функција.

У имплементацији за програмски језик *Wafl* је употребљен други начин. Он је потенцијално ефикаснији, али није то било од пресудног значаја при избору. Основни његов квалитет у контексту истраживачког и експерименталног приступа проблему је у повећаној чистоћи имплементације и олакшаном сагледавању и препознавању евентуалних неисправности како у концептима тако и у имплементацији.

Кориснички дефинисане функције могу да прихвате као аргумент и врате као резултат само типове које подржава *SQL*. Међу њима је само тип ниске довољно флексибилан да се може употребити за преношење произвољних типова програмског језика *Wafl*. Наравно, у истом облику у коме се функцији преноси тело израза – у облику канонизованог текстуалног записа израза.

Евентуални покушај прављења различитих функција за различите типове аргумената и резултата имао би за последицу на стотине различитих функција, а тиме и значајан простор за различите грешке. Такав приступ је далеко од прихватљивог у фази истраживања примењивости концепта.

Имплементација функција

Кориснички дефинисане функције су написане на програмском језику *C*, у складу са правилима писања корисничких функција за СУБП *DB2*. Овде је наведен пример кориснички дефинисане функције `wafl_evaluate_2`, за случај са два аргумента. Функције за

различите бројеве аргумената су имплементиране генерички, помоћу програма на *Wafl*-у и *C++*-у. Ти програми су наведени у додатку *Функције за израчунавање матичних израза за СУБП DB2*.

Ако се узме у обзир да се провера типова програмског кода обавезно изводи пре израчунавања програма, а тиме и пре позивања корисничке функције, онда није неопходно да се у самој функцији додатно експлицитно проверава тип аргумената или резултата. Матични израз, сви аргументи израза, као и резултат израчунавања се преносе у облику текстуалног записа.

```
/*************************************************************
* wafl_evaluate_2
*   Функција израчунава Wafl израз са 2 аргумента
*     улазни:  LONG VARCHAR  code
*     улазни:  LONG VARCHAR  arg1..arg2
*     излазни: LONG VARCHAR  out
*************************************************************/
SQL_API_RC SQL_API_FN wafl_evaluate_2(
    SQLUDF_LONG      *code,         /* улазни:  текст програма      */
    SQLUDF_LONG      *arg1,         /* улазни:  аргумент 1          */
    SQLUDF_LONG      *arg2,         /* улазни:  аргумент 2          */
    SQLUDF_LONG      *out,          /* излазни: резултат као ниска  */
    SQLUDF_NULLIND   *codenull,     /* NULL индикатор               */
    SQLUDF_NULLIND   *arg1null,     /* NULL индикатор               */
    SQLUDF_NULLIND   *arg2null,     /* NULL индикатор               */
    SQLUDF_NULLIND   *outnull,      /* NULL индикатор               */
    SQLUDF_TRAIL_ARGS)              /* пратећи аргументи            */
{
    if( *codenull || *arg1null || *arg2null  ){
        *outnull = -1;
        return 0;
        }
    *outnull = 0;
    if(!waflHandler.hLib){
        setLibError(out);
        return 1;
        }

    WaflHandlerRequest* req = waflHandler.AllocateRequest();
    req->ArgCount = 0;
    req->ProgramCode = terminatedLongVarchar( code );

    req->ArgumentType[req->ArgCount] = WDT_GENERIC;
    req->ArgumentLen[req->ArgCount] = arg1->length;
    req->Argument[req->ArgCount++] = terminatedLongVarchar( arg1 );

    req->ArgumentType[req->ArgCount] = WDT_GENERIC;
    req->ArgumentLen[req->ArgCount] = arg2->length;
    req->Argument[req->ArgCount++] = terminatedLongVarchar( arg2 )

    return processStringResult(req,out);
}
```

Програм 1: Кориснички дефинисана функција за израчунавање дате функције на *Wafl*-у са два дата аргумента – код на програмском језику *C*.

На почетку функције се провера исправност аргумената (у контексту позива функције а не самог матичног израза). Затим се иницијализује захтев за израчунавање матичног израза. Објекат `waflHandler` представља инстанцу интерпретатора за *Wafl*, а објекат `req` класе `WaflHandlerRequest` садржи податке о захтеву за израчунавање. Захтев за израчунавање се попуњава подацима о матичном изразу и аргументима. Израчунавање и обрађивање резултата израчунавања се обављају функцијом `processStringResult`.

Да би се кориснички дефинисана функција могла употребљавати од стране СУБП-а, она се мора направити као објекат базе података. Такође, потребно је дати привилегију за извршавање направљене функције одговарајућим корисницима. Одговарајуће наредбе *SQL*-а су наведене у програму 2. Објекат функције се прави уз претпоставку да је функција имплементирана на програмском језику *C*, као функција `wafl_evaluate_2`, у оквиру библиотеке `waflevaludf.dll`. Путања се не наводи јер систем *DB2* имплицитно прописује где се морају налазити библиотеке са серверским процедурама и кориснички дефинисаним функцијама. Функције се праве у схеми `wafl`, која се експлицитно наводи и при њеној употреби. Свим корисницима базе података се даје привилегија извршавања ове кориснички дефинисане функције:

```
create function wafl.WaflEvaluate( LONG VARCHAR, LONG VARCHAR, LONG VARCHAR )
    returns LONG VARCHAR
    external name 'waflevaludf.dll!wafl_evaluate_2'
    language c
    parameter style db2sql
    not deterministic
    reads sql data
    external action
    fenced;

grant execute on
    function wafl.WaflEvaluate( LONG VARCHAR, LONG VARCHAR, LONG VARCHAR )
    to public;
```

Програм 2: Наредбе *SQL*-а за прављење кориснички дефинисане функције и додељивање права за извршавање свим корисницима.

Имплементација функција за строго типизирано израчунавање

У одељку 10.4, је детаљно описано како се подаци сложених типова чувају у бази података. Уколико се таквим подацима приступа искључиво путем програма написаних на *Wafl*-у, они ће поуздано бити исправно одржавани и вредности атрибута ће у потпуности одговарати декларисаним типовима. Међутим, ако се ти подаци употребљавају из неког другог програмског језика, или у упитима на *SQL*-у у непосредном командном режиму, тада се они могу употребљавати скоро као да су у питању обичне ниске. Очигледна последица је да се на тај начин, уместо исправних података одговарајућег типа, у бази података могу појавити произвољне ниске. Самим тим, може се догодити и да типови података буду неисправни.

Представљена имплементација кориснички дефинисане функције је исправна ако се концепт матичних израза посматра изоловано од других представљених концепата. Међутим, ако се узме у обзир могућност да неки од аргумената ове функције (укључујући чак и сам код који се израчунава) не долазе непосредно из програма на програмском језику *Wafl*, већ из базе података, онда настаје потреба да се при раду са типовима поступа опрезније. У таквим условима, а због потребе да се оствари поуздано типизирано повезивање програмског језика и базе података, пожељно је да при израчунавању матичних израза могу да се динамички проверавају типови аргумената.

Како би се број и сложеност аргумената функције свели на разумну меру, а уз обезбеђивање довољно информација за исправно проверавање усаглашености типова, пожељно је да се избегне преношење типова чија исправност се не доводи у питање. Проблематични су они аргументи који одговарају атрибутима који се читају из базе података. Због тога је потребно да се при позивању функције наведе спецификација

очекиваног типа података за све атрибуте из базе података. Ипак, није практично правити различите врсте функција за сваку могућу комбинацију аргумената чији се типови морају наводити. Следи да је једино прихватљиво решење да се омогући да се за све аргументе експлицитно наводи тип, при чему ће се он заиста наводити за податке који долазе непосредно из базе података, а за остале аргументе ће само бити означено да се ради о неком *Wafl* типу, који се свакако може аутоматски проверити. Пример кода *Програм 3* илуструје начин писања строго типизираних кориснички дефинисаних функција.

```
/*************************************************************
* wafl_evaluate_dt_2
*   функција израчунава Wafl израз са 2 типизирана аргумента
*     улазни:  LONG VARCHAR  code
*     улазни:  LONG VARCHAR  arg1..arg2
*     улазни:  INTEGER  arg1type..arg2type
*     излазни: LONG VARCHAR  out
*************************************************************/
SQL_API_RC SQL_API_FN wafl_evaluate_dt_2(
    SQLUDF_LONG     *code,          /* улазни:  текст програма     */
    SQLUDF_LONG     *arg1,          /* улазни:  аргумент 1         */
    SQLUDF_INTEGER  *arg1type,      /* улазни:  тип аргумента 1    */
    SQLUDF_LONG     *arg2,          /* улазни:  аргумент 2         */
    SQLUDF_INTEGER  *arg2type,      /* улазни:  тип аргумента 2    */
    SQLUDF_LONG     *out,           /* излазни: резултат као ниска */
    SQLUDF_NULLIND  *codenull,      /* NULL индикатор              */
    SQLUDF_NULLIND  *arg1null,      /* NULL индикатор              */
    SQLUDF_NULLIND  *arg1typenull,  /* NULL индикатор              */
    SQLUDF_NULLIND  *arg2null,      /* NULL индикатор              */
    SQLUDF_NULLIND  *arg2typenull,  /* NULL индикатор              */
    SQLUDF_NULLIND  *outnull,       /* NULL индикатор              */
    SQLUDF_TRAIL_ARGS)              /* пратећи аргументи           */
{
    if( *codenull || *arg1null || *arg1typenull
                  || *arg2null || *arg2typenull  ){
        *outnull = -1;
        return 0;
        }
    *outnull = 0;
    if(!waflHandler.hLib){
        setLibError(out);
        return 1;
        }

    WaflHandlerRequest* req = waflHandler.AllocateRequest();
    req->ArgCount = 0;
    req->ProgramCode = terminatedLongVarchar( code );

    addTypedArg( req, arg1, arg1type );
    addTypedArg( req, arg2, arg2type );

    return processStringResult(req,out);
}
```

Програм 3: *Динамички типизирана кориснички дефинисана функција за израчунавање дате функције на Wafl-у са два дата аргумента – код на програмском језику C.*

Основна структура кода је иста као у претходном случају. Детаљи имплементације су наведени у додатку *Функције за израчунавање матичних израза за СУБП DB2*, у облику програма `createWaflEvalUdfCpp.wafl` и `waflevaludf.cpp`, којима се праве све одговарајуће функције.

Модул WaflHandler

За израчунавање *Wafl* израза на страни сервера задужен је модул *WaflHandler*. Овај модул представља пуну инстанцу интерпретатора за програмски језик *Wafl*, која је уоквирена танким позивним слојем. Позивни слој садржи интерфејс путем кога овај интерпретатор могу употребљавати кориснички дефинисане функције.

Интерфејс омогућава све основне услуге које пружа интерпретатор. Основу интерфејса, написаног на програмском језику *C++*, чине структурни тип `WaflHandlerRequest` и функције за рад са њим[43]. Структура типа `WaflHandlerRequest` је носилац размене информација и израчунавања у односу на израчунавање једног програма:

- Тип структуре `WaflHandlerRequest` је:

```
struct WaflHandlerRequest {
    int             MaxArgCount;
    const char*     ProgramCode;
    int             ArgCount;
    char*           ArgumentType;
    const char**    Argument;
    int*            ArgumentLen;
    char            ResultType;
    const char*     Result;
    int             ResultLen;
    bool            ResultState;
    void*           _InternalData;
};
```

- Нови примерак структуре се прави функцијом:

```
WaflHandlerRequest* AllocateRequest();
```

- Припрема структуре се може обавити непосредно или помоћу неколико помоћних функција:

```
void setIntArg( WaflHandlerRequest* req, int argIndex, int n);

void setBoolArg( WaflHandlerRequest* req,
                 int argIndex, bool n );

void setFloatArg( WaflHandlerRequest* req,
                  int argIndex, float x );

void setStringArg( WaflHandlerRequest* req,
                   int argIndex, char* s, int len );

void setGenericArg( WaflHandlerRequest* req,
                    int argIndex, char* s );
```

- Израчунавање вредности израза обавља функција:

```
int HandleRequest( WaflHandlerRequest* );
```

- Прихватање резултата израчунавања се може обавити непосредно или помоћу неке од помоћних функција:

---

[43] Иако је сам интерфејс написан на програмском језику *C++*, интерфејс почива на употреби структура и показивача на функције, како би био у потпуности употребљив и из програма писаних на пограмском језику *C*.

```
int getIntResult( WaflHandlerRequest* req );
bool getBoolResult( WaflHandlerRequest* req );
float getFloatResult( WaflHandlerRequest* req );
```

- За ослобађање непотребног захтева служи функција:

```
void FreeRequest( WaflHandlerRequest* );
```

Начин примене представљених функција се може сагледати кроз имплементацију кориснички дефинисаних функција, у додатку *Функције за израчунавање матичних израза за СУБП DB2*.

Начин превођења матичних израза

Превођење матичних израза је тесно везано за проверавање типова упитних израза са матичним изразима. Најсложенији део превођења је управо анализа матичног израза и поступак распознавања и проверавања типова. Алгоритам распознавања типова матичних израза у програмском језику *Wafl* је наведен у поглављу 12.

11.5. Библиотеке кода у бази података

Имплементација библиотека програмског кода у бази података се не разликује много од имплементације библиотека у датотекама. Једина значајна разлика је што се дефиниције елемената библиотеке не читају из датотеке него из базе података. Када се текст дефиниција прочита, начин превођења је практично исти.

Значајна последица употребе библиотека у бази података јесте да у случају програма, који употребљавају такве библиотеке, већ у фази превођења програма мора да се остварује повезивање са базом података. У контексту програмског језика *Wafl* то не представља велику новину, јер се такво повезивање већ мора остваривати ради провере типова упитних израза.

12. Имплементација провере типова упитних израза

Алгоритми за проверавање типова упитних функција и матичних израза, који су представљени у поглављу 9, могу се применити и на програмски језик *Wafl* уз минимална прилагођавања. Неопходне измене се односе на специфичности имплементације размене информација о сложеним типовима и начина извршавања матичних израза на страни СУБП-а.

Распознавање типа упитне функције у програмском језику Wafl

1. Ако упитна функција има аргументе, онда:
   1.1 Свако појављивање матичне променљиве у тексту упита се замењује јединственом ознаком матичног параметра, при чему се прави таблица пресликавања матичних параметара у одговарајуће параметре;
2. Управљачки модул тражи од СУБП-а да препозна тип резултата упита, као и да на основу места на коме се употребљавају распозна неопходне типове матичних параметара;
3. Ако СУБП успе да препозна све тражене типове, онда:
   3.1 Ако међу типовима које је СУБП препознао постоји неки од кориснички дефинисаних типова, који су наведени у табели `wafl.dtypes`, онда управљачки модул замењује сваки такав тип одговарајућим типом програмског језика *Wafl*;
   3.2 Ако се разликују типови параметара који се односе на једну исту матичну променљиву:
      3.2.1 Покушава се њихова унификација;
      3.2.2 Ако унификација не успе, значи да постоје неисправности у упиту:
         3.2.2.1. Прекида се провера типа;
         3.2.2.2. Преводилац се обавештава о проблему;
   3.3 Од препознатих типова матичних променљивих и резултата се прави одговарајући функцијски тип и прослеђује се преводиоцу као тип упитне функције, како би био искоришћен у даљем проверавању типова у програму;
4. У супротном, постоје неисправности у упиту:
   4.1 Прекида се провера типа;
   4.2 Преводилац се обавештава о проблему.

Алгоритам 11: Препознавање типа упитне функције у *Wafl*-у

Распознавање типа матичног израза у програмском језику Wafl

1. Препознају се имена у матичном изразу која се односе на аргументе упитне функције (тј. матичне променљиве);
2. Препознају се имена у матичном изразу која би могла се односе на атрибуте базе података који се издвајају самим упитом на *SQL*-у[44];
3. Помоћу СУБП-а се покуша распознавање имена атрибута и њихових типова:
   3.1 Ако се у упиту непосредно употребљавају неке матичне променљиве, онда:
      3.1.1 Свако појављивање матичне променљиве у тексту упита се замењује јединственом ознаком параметра, при чему се води таблица пресликавања матичних параметара у одговарајуће параметре;
   3.2 Упит на *SQL*-у се привремено трансформише тако да се сва имена која би могла да се односе на атрибуте базе података експлицитно додају у клаузулу `SELECT`, као нови атрибути који се читају упитом;
   3.3 Сваки матични израз у упиту се привремено замењује по једном новом матичном променљивом, тј. одговарајућим параметром;
   3.4 Посредством управљачког модула базе података се изврши припремање упита и распознавање типова од стране СУБП-а;
   3.5 Ако је дошло до грешке у припреми упита, онда се поступак прекида и извештава се о неисправностима;
   3.6 Ако међу типовима које је СУБП препознао постоји неки од кориснички дефинисаних типова, који су наведени у табели `wafl.dtypes`, онда управљачки модул замењује сваки такав тип одговарајућим типом програмског језика *Wafl*;
   3.7 Ако се разликују типови параметара који се односе на једну исту матичну променљиву:
      3.7.1 Покушава се њихова унификација;
      3.7.2 Ако унификација не успе, значи да постоје неисправности у упиту:
         3.7.2.1. Поступак се прекида и извештава се о неисправностима;
   3.8 Установљавају се типови прочитаних атрибута и то како оних који се читају у оригиналном упиту, тако и оних који су додати у кораку 3.1.1;
4. Сваки матични израз који употребљава аргументе упитне функције и/или атрибуте базе података се трансформише у канонски облик примене ламбда функције на одговарајуће аргументе и атрибуте;
5. Сваком матичном изразу се проверава тип:
   5.1 Проверава се тип ламбда функције која одговара изразу;
   5.2 Проверава се усаглашеност типа ламбда функције са:
      5.2.1 типовима аргумената упитне функције (који могу бити већ установљени у кораку 3, ако се непосредно употребљавају као матичне променљиве);
      5.2.2 типовима употребљаваних атрибута базе података који се преносе тој функцији (који се добијају кораком 3);
      5.2.3 очекиваним типом резултата матичног израза (који се добија кораком 3, у облику типа матичне променљиве којом је израз привремено замењен);
6. Проверава се међусобна усаглашеност типова матичних израза:
   6.1 У дефиницију упитне функције се привремено додају поддефиниције које одговарају матичним изразима:
      6.1.1 свака поддефиниција представља делимичну примену одговарајуће ламбда функције на аргументе упитне функције;

[44] Као што је већ наглашено у опису синтаксе и семантике матичних израза у програмском језику *Wafl*, најпре се препознају сва имена програмског језика *Wafl*, па затим имена атрибута базе података. У складу са тиме се у овом кораку за сва имена, која се не препознају као имена из *Wafl* кода или аргументи упитне функције, претпоставља да се односе на атрибуте базе података.

6.2 Проверава се усаглашеност типова упитне функције и одговарајућих поддефиниција;

7. У упиту на *SQL*-у се сваки матични израз замењује позивом одговарајуће кориснички дефинисане функције за извршавање програма на *Wafl*-у, при чему се као аргументи те функције наводе:
   7.1 сам матични израз (у облику одговарајуће ламбда функције);
   7.2 матичне променљиве које представљају референце на аргументе упитне функције, који се користе у матичном изразу;
   7.3 атрибути који потичу из упита а користе се у матичном изразу;
8. По потреби се обезбеђују експлицитне конверзије типова аргумената и резултата позива кориснички дефинисане функције;
9. Тако трансформисан упит још једанпут пролази кроз проверу типова од стране СУБП-а ради препознавања одређених спорних случајева, које би при израчунавању могле да доведу до грешака чак и када је провера по елементима упита прошла успешно;
10. Ако су сви претходни кораци успешно окончани, тада је провера типова успела и типови су међусобно усаглашени. У супротном, ако у било ком кораку дође до грешке, поступак се прекида и извештава се о неисправностима.

Алгоритам 12: Проверавање типа матичних израза у *Wafl*-у

13. Примене

Представљена имплементација повезивања програмског језика *Wafl* и РСУБП-а *IBM DB2* је примењена у више различитих области. Поред развоја Веб апликација имплементација је употребљавана у области управљања базама података и научног израчунавања.

Најзначајнији примери развијених Веб апликација су Уџбеник из аналитичке геометрије [Miti2003], Библиотека новинских исечака Музеја Николе Тесле [Malk2008a] и Библиотека Громанових фотографија [Malk2008a]. У свакој од наведених апликација се дословно сви подаци чувају у бази података. Сви динамички садржаји су аутоматски израчунавани на основу тако сачуваних података, тако да ове Веб апликације практично уопште не употребљавају систем датотека[45]. Због тога су до пуног изражаја дошле добре стране обједињеног трансакционог простора, а нису се исказале слабости које су последица тога што у представљеној имплементацији трансакциони простор не обухвата систем датотека.

У оквиру пројекта развоја информационог система Математичког факултета [Malk2007], представљено решење је примењено за аутоматизацију неких послова управљања базом података. Најважнији тако аутоматизовани послови су прављење и одржавање табела и окидача за аутоматско праћење промена и прављење дијаграма релација базе података. У тим пословима је дошла до изражаја флексибилност и апстрактност функционалног програмског језика. У имплементацији различитих делова ових послова су употребљавани како строго типизиран тако и слабо типизиран (генерички) начин рада са базом података.

Примене у области научног израчунавања су се одвијале у оквиру биоинформатичких истраживања везаних за секундарну структуру протеина [Živk2006], [Malk2008c], [Malk2009]. Углавном се ради о претраживањима података и различитим статистичким израчунавањима. За разлику од примене на Вебу, у овим случајевима није била потребна трансакциона обрада, али су употребљавани матични изрази. Њихова примена је утицала на поједностављивање писања програма за анализу података.

У оквиру истог истраживачког пројекта, али у вези са другим проблемима, појавила се потреба за већом процесорског снагом, због чега су вршени експерименти са дистрибуираним израчунавањем. Анализа и дискусија могућности примене дистрибу-

[45] Осим што су саме апликације записане у облику датотека.

ираног израчунавања помоћу представљеног начина повезивања програмских језика и база података је наведена у одељку 14.7. Једноставна имплементација библиотеке за дистрибуирано израчунавање је приложена у додатку *Пример имплементације подсистема за дистрибуирано израчунавање*, на страни 193.

Општи утисак након описаних примена је веома позитиван. Показало се да је добро што се имплементација одликује високим нивоом флексибилности у односу на начин повезивања са базом података. У неким случајевима је била пресудна могућност употребе генеричког приступа садржају базе података, док је у другим случајевима било веома корисно строго типизирано повезивање. Слично томе, док се у развоју Веб апликација махом употребљавао статички приступ, у пословима аутоматизације управљања базом података често је било потребно аутоматски правити и израчунавати сложене упитне изразе и наредбе, што је било могуће једино ослањањем на динамички приступ подацима.

Сваки од предложених концепата је потврђен у некој од остварених примена. Употреба упитног језика у програмском језику и строга типизираност повезивања су имали значајну улогу у свим наведеним применама. Употреба програмског језика у упитном језику, у облику матичних израза, је допринела унапређивању решења у области анализа података. Јединствен трансакциони простор је употребљаван у случају Веб апликација, где су у потпуности потврђени значај и употребљивости тог концепта, чак и у случају његове непотпуне имеплементација (тј. без обухватања система датотека и других рачунарских ресурса). Ортогоналност у односу на типове података је омогућила ширину примене, при чему је записивање израза програмског језика и програма у бази података примењено у експериментима са дистрибуираним израчунавањем. Потреба за ортогоналношћу у односу на начине повезивања је потврђена кроз употребу различитих начина повезивања у оствареним применама.

Практична примена је омогућила да се уоче и отклоне неке слабости у повезивању, што је допринело да концепти повезивања и одговарајућа имплементација добију облик у коме су овде представљени. Штавише, због неких уочених проблема је унапређиван и сам програмски језик *Wafl*. На пример, након што је уочено да се често употребљава делимично израчунавање ламбда израза, уведен је облик ламбда израза са додатним везивањем имена (одељак 4.1.2).

Део IV

Дискусија

14. Дискусија

14.1. Матични изрази

14.1.1. Особине имплементације матичних израза

Матичних изрази имају широк домен примењивости. Омогућавају повећавање изражајности кода и једноставније и чистије писање кода којим се решавају проблеми који се не могу решити само применом упитног језика.

Сам концепт матичних израза не поставља никаква ограничења у односу на могућности примене матичних израза. Као што је већ наглашено, матични изрази се могу примењивати у изразима упитног језика на свим местима на којима синтакса допушта употребу литерала, било да се ради о скалару, скупу скалара или табели. Међутим, одређена ограничења уводе имплементације СУБП-а и програмског језика. На пример, *DB2* не допушта позивање кориснички дефинисаних функција на свим местима на којима се могу наћи литерали, па у таквим случајевима није могућа употреба матичних израза имплементираних на описан начин. Друго значајно ограничење је да није могуће да се кориснички дефинисана функција (а тиме ни матични изрази) употребљава као логички израз упитног језика, ни у оквиру услова рестрикције ни на другим местима.

Експериментална имплементација матичних израза у програмском језику *Wafl* има неколико ограничења:

- подржани су само скаларни матични изрази;
- није могуће употребљавати матичне изразе у подупитима;
- ако је резултат матичног израза сложен тип, бази података се враћа тип `String` и
- употреба упита у самим матичним изразима је ограничена.

Ограничавање на скаларне матичне изразе значи да се матични изрази могу употребљавати у упитима само на местима на којима се очекује скалар. Са друге стране, вредност матичног израза може имати било који тип подржан у програмском језику.

У случају употребе матичних израза у подупитима, проблем је у размени информација о типовима елемената подупита, због чега је неопходно увести додатни ниво

интеграције са СУБП-ом, што у овој фази није било могуће. Проблем је и у самом концепту подршке за сложене типове, која почива на сарадњи СУБП-а и програмског језика, због тога што би резултат матичног израза сложеног типа у подупиту могао да се у главном упиту употребљава као ниска, што нарушава оквире строге типизираности. Наиме, због тога што се матични изрази израчунавају функцијама СУБП-а, оне морају да израчунавају резултат неког типа који СУБП може да препозна. У случају да је резултат неког сложеног типа, он се преноси у канонизованом текстуалном облику.

Један од значајних доприноса концепта матичних израза је у могућности да се део израчунавања пренесе на сервер без потребе да се предузима релативно сложен развој специфичних кориснички дефинисаних функција. На тај начин се подаци употребљавају на месту на коме се и налазе, чиме се остварује уштеда у мрежним ресурсима и омогућава централизација процесорских капацитета. Ова тема је детаљније обрађена у одељку 14.7. Пример 17 илуструје примену матичног израза у оквиру колонске функције упитног језика. Упит рачуна највећу суму елемената листе записане у оквиру атрибута `list` табеле `intlists`. Сума елемената сваке од листа се рачуна на страни сервера, у оквиру израчунавања упита. Ако би се одговарајуће израчунавање имплементирало на страни клијента, било би неопходно да се све листе из ове табеле прочитају и пренесу до клијентског програма.

```
q()
where{
    q() = typed sql query {
        select max( :(sum(list)) )
        from intlists
        };

    sum(l) =
        if l->empty() then 0 else l->hd() + l->tl()->sum();
}
```

Пример 17: Примена матичних израза у изразима на *SQL*-у

Пример 18 илуструје примену матичног израза у оквиру услова рестрикције. Упит издваја из базе података само оне листе чија је сума елемената већа од дате вредности. Слично, матични изрази се могу употребљавати и у условима спајања. Услов рестрикције није могуће записати у облику:

```
where :(sum(list)>n)
```

због наведеног ограничења да се матични изрази не могу употребљавати као логички изрази упитног језика.

```
q(10)
where{
    q(n) = typed sql query {
        select id, list
        from intlists
        where :(sum(list)) > :n
        };

    sum(l) =
        if l->empty() then 0 else l->hd() + l->tl()->sum();
    }
```

Пример 18: Примена матичних израза у клаузули *WHERE* језика *SQL*

Флексибилност и могућности концепта матичних израза се на веома илустративан начин могу сагледати кроз могућност да се у матичним изразима употребљавају изрази упитног језика. Једино ограничење које сам концепт матичних израза поставља у том смислу је сасвим природно – сви такви приступи бази података морају да се одвијају у оквиру исте трансакције.

На пример, функција `studenti()` у програму *Пример 19* чита податке о студентима, укључујући и листу назива свих положених предмета, користећи при томе упитну функцију `ispiti` унутар матичног израза:

```
studenti() = typed sql query {
    select indeks, ime, prezime,
           :( ispiti(indeks)->map(\r:r$NAZIV )) as predmeti
    from student
    };

ispiti(indeks) = typed sql query {
    select naziv, ocena, nastavnik
    from ispit join predmet
        on ispit.id_predmeta = predmet.id
    where indeks = :indeks
    };
```

Пример 19: Употреба упита у матичним изразима

14.1.2. Оптимизација израчунавања матичних израза

При израчунавању матичних израза простор за оптимизацију се може тражити у оквиру самог израчунавања, али и у оквиру преношења израза серверу и припреме израза за извршавање.

Ако се има у виду да се један исти матични израз може израчунавати више пута у једном упиту, као и да се један исти упит може извршавати више пута, очигледно је да се значајна уштеда може остварити увођењем подршке за једнократно припремање израза за извршавање. Један пут за остваривање такве уштеде је да се у оквиру имплементације модула за извршавање матичних израза имплементира складиште за чување преведених матичних израза. Овај вид уштеде може да буде веома значајан, посебно у случају израза чије израчунавање је једноставно, јер је тада већи релативни значај утрошка ресурса на припрему за извршавање.

У случају система *DB2* постоји додатни простор за оптимизацију израчунавања путем једнократног превођења израза. Кориснички дефинисане функције се могу обликовати тако да се преносе подаци између више позива у оквиру израчунавања истог упита. Таква могућност је изворно замишљена за омогућавање имплементирања колонских функција, али се може употребити за преношење преведеног матичног израза. Ово нема непосредне везе са оптимизацијом упита. Додатне оптимизације би се могле остварити ако би се на страни СУБП-а чувале преведене верзије библиотека.

Други начин оптимизације је да се сам поступак припреме израза за извршавање на страни СУБП-а учини ефикаснијим тако што ће се матични изрази предавати СУБП-у у преведеном облику, а не у облику текстуалног записа. Ефекат оваквог вида оптимизације је посебно значајан у случају израза који се израчунавају мали број пута. Како то није уобичајен случај када се ради о базама података, може се закључити да је овакав начин оптимизације мање значајан од претходног. Уобичајен рад са великим количинама

података и вишеструко поновљено извршавање истих упита (а тиме и матичних израза) има за последицу да број понављања припремања и извршавања израза има далеко већи утицај на перформансе од начина њиховог преношења.

Овде није узето у обзир да се програмски код матичног израза у сваком случају мора на неки начин разменити између клијента и сервера. Однос величина текстуалног записа и величине преведеног кода је у великој мери зависан од имплементације, па га није могуће укључити у уопштено разматрање. Такође, ако се програмски језик имплементира, онда се уместо текста програма евентуално може преносити само неки међукод, што може даље умањити значај овакве оптимизације.

14.1.3. Проширивање концепта матичних израза

Представљен концепт матичних израза је флексибилан и може се и даље проширивати. Један од могућих смерова проширивања би могло да буде увођење подршке за колонске матичне изразе. Вредност колонских матичних израза би се израчунавала над свим вредностима неког атрибута (колоне), као у случају колонских функција у *SQL*-у. При дефинисању синтаксе оваквих израза је потребно узети у обзир њихову потенцијално сложену семантику.

Ако се претпостави да СУБП може да подржава израчунавање матичних израза на више различитих програмских језика, онда има смисла разматрати и декларисање језика матичног израза, на пример:

```
... :[Wafl](...) ...
```

Посебан смер проширивања би било увођење посебних матичних израза који би се израчунавали на страни клијента, пре или после израчунавања упитне функције. Израчунавање матичних израза на страни клијента пре израчунавања упитне функције има смисла на местима на којима се вредност матичног израза чита при извршавању наредбе упитног језика. Са друге стране, израчунавање матичних израза на страни клијента после израчунавања упита има смисла на местима на којима се подаци преносе од базе података програму, тј. у у оквиру клаузуле *SELECT* упитних израза. У оба случаја се не ради о суштински новој функционалности, већ само о обогаћивању синтаксе, јер би свака упитна функција са таквим матичним изразима могла да се имплементира и без такве подршке и то на два начина:

- помоћу матичних израза који се израчунавају на серверу – али уз потенцијално смањену ефикасност јер би се у неким случајевима без потребе понављала израчунавања израза који би се могли израчунати једнократно пре израчунавања упита или
- као композиција двеју или више функција – што захтева раздвајање целина у више функција.

Подршка са матичне изразе на страни клијента би могла да се оствари, на пример, у складу са постојећим клијентским матичним изразима у интерфејсу *SQLJ* за повезивање програмског језика *Java* са базама података [Ora:2002]. Као што је већ наглашено, *SQLJ* омогућава израчунавање матичних израза искључиво на страни клијента. Пре самог матичног израза се наводи спецификација да ли се израз израчунава пре (`IN`) или после (`OUT`) наредбе упитног језика:

```
... :IN(...) ...
... :OUT(...) ...
```

14.2. Обрада података сложеног типа

При обликовању концепата повезивања програмских језика и база података, као ни при описивању експерименталне имплементације, није експлицитно наведено на који се начин могу обрађивати подаци сложених типова на страни СУБП-а. Нису представљени никакви специфични механизми који су намењени управо за омогућавање такве врсте обраде података. Сложени типови података се на страни СУБП-а имплементирају генерички. Због тога СУБП нема неопходне информације о унутрашњој структури таквих података, па се елементима унутрашње структуре сложених података не може приступати уз примену постојећих механизама СУБП-а или упитног језика.

Међутим, обрада података сложеног типа јесте могућа захваљујући матичним изразима. Матични изрази се извршавају на страни сервера, при чему се начелно могу налазити у свим деловима упита. У оквиру матичних израза се подаци сложеног типа могу употребљавати са стварним типом, а не у генеричком облику. Да би то било могуће, потребно је само да се за конкретан сложени тип обезбеди пресликавање типова. Пресликавање типова у експерименталној имплементацији повезивања програмског језика *Wafl* и база података је описано у одељку 11.3.

Пример 20 илуструје како се елементи листе могу обрађивати на страни базе података. Упитом се издвајају непарни елементи листе и сума свих елемената листе. У овом случају се у матичном изразу вредност атрибута `list` табеле `intlists` употребљава у складу са употребом података типа `List[Integer]` у програмском језику *Wafl*.

```
qListe() = typed sql query {
    select
        id,
        :( list->filter(\el: el%2 = 1) ) neparni,
        :( sum(list)) sumaSvih
    from intlists
    }
where{
    sum(l) =
        if l->empty() then 0
        else l->hd() + sum(l->tl());
};
```

Пример 20: Обрада података сложених типова на страни СУБП-а

14.3. Јединственост трансакционог простора

Трансакције у програмском језику *Wafl* се односе не само на трајне податке у бази података, већ и на друге трајне податке. Као што је већ наглашено, трансакциони простор експерименталне имплементације обухвата базу података и податке у оквиру апликативне сесије, али не и датотеке. Пуна имплементација, као и неки други језици или разноврснија развојна окружења би требало да обухвате и датотеке, а могли би да употребљавају и неке друге врсте трајних података.

Постојање различитих врста трајних података има за последицу да се њима приступа посредством различитих програмских система, па се овде ради о једном виду окружења са дистрибуираним подацима. Због тога је потребно да имплементирани подсистем за управљање трансакцијама омогући вид управљања дистрибуираним трансакцијама [Rama2002].

При томе се мора имати у виду различитост природе ових трајних података:

- Подацима похрањеним у бази података се приступа посредством СУБП-а, који обухвата и одговарајући подсистем за трансакције. Већи број система обухвата и подсистем за двофазно потврђивање трансакција. Сваки СУБП обезбеђује све четири основне карактеристике трансакција.
- Датотекама се приступа посредством система за управљање датотекама, који је најчешће саставни део оперативног система. Већина система за управљање датотекама не подржава трансакције, углавном због тога што нису у стању да обезбеде атомичност и конзистентност. Трајност је углавном испоштована. Изолованост је испоштована само делимично, при чему основни проблем представља сасвим ограничена грануларност, јер се изолованост обично остварује закључавањем читавих датотека.
- Подацима апликативне сесије се приступа посредством подсистема за управљање параметрима сесије. Како тај подсистем представља саставни део имплементације програмског језика (или одговарајућег развојног окружења), потребно је да се обликује тако да подржава трансакције и двофазно потврђивање трансакција.

Као што је већ наглашено у одељку 11.2, у случају програмског језика *Wafl* није имплементирано двофазно потврђивање трансакција, као ни пуна јединственост трансакционог простора, јер није обухваћен систем датотека.

14.4. Параметризовани уместо динамичких упита

При развоју софтвера су чести случајеви у којима се у зависности од неких параметара бира једна од више варијанти упита, а затим се употребљава у различитим контекстима. У таквим случајевима је уобичајена употреба динамичког *SQL*-а, при чему се различити упити могу преносити у облику ниски, као записи упита на упитном језику.

У случају програмског језика *Wafl*, упаривањем строге типизираности и делимичног израчунавања, добија се нов квалитет у раду са базама података. Делимично израчунавање омогућава да се, на сасвим једноставан начин, од упитних функција са више аргумената праве нове упитне функције са мање аргумената. Строга типизираност, са друге стране, обезбеђује да се типови различитих упитних функција установљавају и упоређују већ у фази превођења програма, чиме се онемогућава појављивање грешака у типовима података у фази извршавања програма, које су карактеристичне за динамички *SQL*. При свему томе, захваљујући функционалној природи језика, упитима се може руковати у облику функција, тј. неизрачунатих и параметризованих упита.

Пример 21 садржи *Wafl* код, у којем се строго типизирани статички упити употребљавају уместо динамичког SQL-а. Предности у односу на уобичајена решења су вишеструке. Одабир упита се одвија независно од његове примене, при чему се, применом делимичног израчунавања, у упитима који имају више аргумената везују они аргументи који су већ познати. Захваљујући томе се параметри, на основу којих се врши одабир упита, не морају преносити дубоко кроз програм, све до места употребе упита. Такође, и сам код за одабир упита се наводи само једанпут. Ова флексибилност је последица нивоа апстрактности, којим се одликују *Wafl* и други функционални програмски језици.

```
...
 (
  if smer > 0
    then qStudentiSmera(smer,_)
  else if godina > 0
    then qStudentiGodine(godina,_)
  else qSviStudenti
)
->sviIzvestaji( imena )
...
where{
    sviIzvestaji( upit, imena ) =
        imena
        ->map( jedanIzvestaj(upit,_,redIzvestaja) )
        ->strJoin('\r\n----------\r\n');

    qSviStudenti(ime) = typed sql query {
        select * from db.dosije
        where ime like '%'||:ime||'%'
        };

    qStudentiSmera(smer,ime) = typed sql query {
        select * from db.dosije
        where id_smera = :smer
          and ime like '%'||:ime||'%'
        };

    qStudentiGodine(godina,ime) = typed sql query {
        select * from db.dosije
        where indeks/10000 = :godina
          and ime like '%'||:ime||'%'
        };
    ...
}
```

Пример 21: Пример употребе строго типизираних статичких упита у случају када избор упита зависи од неких параметара

14.5. Неизрачунати изрази у бази података

Подршка за записивање података сложених типова у бази података је довољно општа да омогућава записивање произвољног нормализованог израза. Једино ограничење представља вредна природа програмског језика *Wafl*, која налаже да се сваки аргумент неке функције или операције израчуна пре његовог преношења. Тиме је онемогућено да се у бази података непосредно запишу неизрачунати изрази. Записивање неизрачунатих израза може имати смисла у неким специфичним случајевима. У неке од потенцијалних примена спадају прављење аутоматских тестова или туторијала, али и дистрибуирање израза који би се израчунавали на другом рачунару.

Пример 22 илуструје два начина да се превазиђе ово ограничење. Један начин је да се изрази записују у облику функција без аргумената. Функција без аргумената је нормализован израз, па се може записати у бази података. По читању из базе података се може по потреби применити или употребљавати у неизмењеном облику. Такав начин рада је строго типизиран, при чему се уместо вредности израза типа *T* записује одговарајућа функција без аргумената типа (->*T*).

```
// записивање вредности израза у бази података
    (<израз>)->zapiši(...)
// читање вредности израза из базе података
    ...pročitaj(...)...

// записивање израза у бази података у облику функције без аргумената
    (\:<израз>)->zapiši(...)
// читање израза из базе података у облику функције без аргумената
    ...pročitaj(...)()...

// записивање израза у бази података помоћу мета-функција
    (<израз>)->metaGetSource()->zapiši(...)
// читање израза из базе података помоћу мета-функција
    ...pročitaj(...)->metaEvaluate()...
    ...pročitaj(...)->metaCompile()...
```

Пример 22: Неизрачунати изрази у бази података

Други начин је да се употребљава библиотека мета-функција. У програмском језику *Wafl* постоји библиотека мета-функција, чија изворна намена нема никакве везе са базама података и темом овог истраживања. Ове функције су намењене за различита израчунавања над програмским кодом, у циљу писања програма за различите анализе програма на *Wafl*-у, а у склопу истраживања неких проблема у области семантике и оптимизације. У овом контексту су важне три функције те библиотеке: `metaGetSource`, `metaCompile` и `metaEvaluate`.

Функција `metaGetSource` је полиморфна унарна функција типа `('1->String)`, која за аргумент било ког типа израчунава ниску која представља његов канонизован изворни код. Значајна особина ове функције је њено специфично понашање у односу на аргумент. Да би се могао израчунати изворни код произвољног израза, па и оног који није нормализован, ова функција се дефинише са карактеристичном лењом семантиком. По томе је различита од свих осталих функција у програмском језику *Wafl*.

Семантика ове функције је веома слична семантици поступка аутоматског канонизовања израза. Штавише, у разматраној имплементацији програмског језика *Wafl* њихове имплементације почивају на истом коду. Једина разлика је у томе што се пре самог прослеђивања неког израза бази података тај израз увек најпре израчуна, па тек онда канонизује, док функција `metaGetSource` има општије понашање, јер се израчунава над изразом који представља њен аргумент, *без претходног израчунавања тог аргумента*. На тај начин се омогућава да се у израчунавању, па и у записивању у бази података, употребљава канонизован текстуални запис неизрачунатог израза[46].

[46] Библиотека мета-функција не садржи функцију која израчунава канонизован запис израчунатог израза. Таква функција се, у случају потребе, може сасвим једноставно дефинисати, захваљујући томе што све функције написане на *Wafl*-у увек имају вредну семантику:

```
metaGetSourceX(x) = metaGetSource(x)
```

За израчунавање тако записаног израза се може употребити унарна функција `metaEvaluate`, типа `(String->String)`. Она израчунава израз дат у облику текстуалног записа. Резултат ове функције је ниска која представља резултат израчунавања израза, у облику канонизованог текстуалног записа.

Трећа функција, `metaCompile`, је слична функцији `metaEvaluate`, али има полиморфан тип `(String->'1)`. Она преводи текстуално записан израз на програмском језику *Wafl* и затим га израчунава, при чему се резултат не канонизује већ остаје у свом основном облику. Њен тип имплицира да ова функција није строго типизирана, јер омогућава да се она преведе у дословно сваком контексту, уз претпоставку да „програмер зна шта ради“, што у одређеним условима може да има за последицу типовне грешке у фази извршавања[47].

14.6. Библиотеке кода у бази података

Библиотеке кода у бази података могу имати вишеструку примену. Три могуће примене се издвајају као посебно значајне:

- проширивање скупа функција које могу да се користе, а чији код не мора да се наводи у програму;
- централизовано инсталирање кода и
- имплементација делова кода који су везани за структуру базе података.

Прва примена је уједно и основна намена свих врста библиотека. У том смислу, библиотеке кода у бази података су специфичне само по форми. По практичним аспектима употребе нема значајних разлика у односу на уобичајене датотечне библиотеке. Специфичност библиотека у бази података се више огледа у другим могућим применама.

Проблем синхронизованог инсталирања програмског кода на клијентске рачунаре је један од најсложенијих послова у великим рачунарским окружењима са великим бројем радних станица и распрострањеним клијентским апликацијама. Дистрибуирањем кода у виду садржаја базе података обезбеђује се централизовано место са кога апликације динамички преузимају и употребљавају делове програмског кода. Штавише, на такав начин се могу дистрибуирати и читаве апликације. Пример 23 показује како се клијентски програм може дефинисати као школка која позива програм записан у бази података.

```
dbLib::program()
where {
    dbLib = library db 'testApp';
};
```

Пример 23: Пример клијентске апликације чији се програмски код дистрибуира посредством библиотеке кода у бази података.

[47] При имплементацији ове функције се води рачуна о очекиваном типу преведеног и израчунатог израза (који је установљен при провери типова) што омогућава да се неке грешке избегну применом имплицитних конверзија, али у већини случајева то ипак није могуће. На пример, функција са два аргумента се не може применити на три аргумента.

Веома значајна могућа примена се односи на имплементацију делова кода који су релативно тесно спрегнути са структуром базе података. Ако би се функције, које остварују непосредан приступ бази података, написале као саставни делови библиотека похрањених у бази података, тиме би се постигло да делови кода који зависе од конкретне базе података буду записани управо у тој истој бази података, што омогућава модуларнији развој и дистрибуцију делова кода који су тесно спрегнути са структуром или садржајем базе података.

14.7. Мрежно израчунавање

Могућности матичних израза и записивања сложених података у бази података отварају простор за примену представљеног интегрисаног решења у домену мрежног израчунавања. Термин „мрежно израчунавање“ се овде користи као уопштење различитих начина израчунавања делова програма на различитим рачунарима који су повезани у мрежу. У наредним одељцима су анализиране могућности примене представљеног решења у доменима дистрибуираног и централизованог израчунавања.

14.7.1. Дистрибуирано израчунавање

Једна од значајних могућих примена представљеног концепта интегрисања програмског језика и базе података је у области дистрибуираног израчунавања. Под дистрибуираним израчунавањем се подразумева да се различити делови (задаци, послови) једног сложеног израчунавања одвијају на различитим рачунарима.

Да би се неки послови могли дистрибуирати на више рачунарских система, неопходно је успоставити одговарајућу комуникацију између тих система. У случају примене представљеног интегрисаног решења, специфичност се огледа у томе да се у свим фазама дистрибуираног израчунавања комуникација може остваривати кроз базу података:

- *Дистрибуирање програма који управљају израчунавањем* се може остварити коришћењем базе података као библиотеке програмског кода. Тиме се одржавање тих система своди на најмању могућу меру.
- *Задаци се могу поднети на распоређивање* записивањем у бази података. Како се у бази података може записати било који програм на матичном језику, могу се записати и задаци које је потребно израчунавати на различитим рачунарским системима.
- *Подаци који се употребљавају у израчунавању* се такође могу дистрибуирати кроз базу података.
- *Координирање расподеле задатака* се може изводити уз евидентирање свих значајних информација у бази података. На тај начин све одговарајуће информације постају доступне свима који учествују у израчунавању. Једна од последица је да се праћење израчунавања може одвијати на било ком систему који може приступати бази података, па чак и када на њему не постоје компоненте које се односе на конкретан програмски језик и постављање или израчунавање задатака.

- *Прикупљање резултата израчунавања* се такође може изводити применом механизама базе података.

Ради тестирања концепта у пракси и сагледавања евентуалних ограничења, имплементиран је сасвим једноставан модел за подршку дистрибуираног израчунавања. Написани су програми `run.wafl`, који има улогу агента који израчунава направљене задатке, и програм `dist.wafl`, који представља пример клијентског програма који прави задатке и прикупља резултате израчунавања. Заједнички делови кода су обухваћени библиотеком `distributed.wlib`.

Дистрибуирање израчунавања се обавља функцијом `distMap(lst,fn)`. Резултат израчунавања ове функције је еквивалентан резултату израчунавања функције `map(lst,fn)` – представља листу чији су елементи резултати примене дате функције `fn` на елементе листе `lst`, уз очување редоследа. Разлика је у томе што се функција `fn` не израчунава на месту израчунавања програма већ се њено израчунавање дистрибуира.

Дистрибуирање послова се састоји од прављења нових послова и прикупљања њихових резултата. Функција `doCreateJob(fn,x)` прави нови посао, који ће израчунавати `fn(x)`, и израчунава као резултат јединствен идентификатор направљеног посла, а функција `waitJobResult(id)` чека да се заврши израчунавање посла са датим идентификатором и израчунава резултат посла.

```
distMap( lst, fn ) = lst->map( doCreateJob(fn,_) )
                       ->map( waitJobResult );
```

Пуне имплементације функција `distMap`, `doCreateJob` и `waitJobResult` садржане су у оквиру библиотеке `distributed.wlib`, која је заједно са програмима `run.wafl` и `dist.wafl` и резултатима тестирања ефикасности примене библиотеке приложена у додатку *Пример имплементације подсистема за дистрибуирано израчунавање*. Значајна карактеристика приложене библиотеке је строга типизираност.

14.7.2. Централизовано израчунавање

Као супротност концепту дистрибуираног израчунавања може се разматрати концепт централизованог израчунавања. Представљен концепт интегрисања програмског језика и база података може се применити и за имплементирање централизованог израчунавања.

Омогућавањем израчунавања матичних израза у оквиру упита створене су неопходне претпоставке да се систем за управљање базама података употребљава и као апликативни сервер, који би преузео на себе послове израчунавања делова програма. При томе клијентски и серверски делови кода могу да употребљавају исте потпрограме, чиме се развој значајно поједностављује. Уједначавање метода развоја има за последицу да се промена места израчунавања делова кода може изводити на једноставнији начин, јер није неопходно писати исти код на различитим језицима и за различите архитектуре. Значајну улогу при томе има строга типизираност кода, којом се добија додатна поузданост решења.

У примерима кода *Пример 24* и *Пример 25* се израчунава функција `f` за вредности атрибута `X` које се читају из базе података. У првом случају се израчунавање одвија на страни клијента, а у другом на страни сервера. Примери илуструју како се сасвим једно-

ставно може променити место извршавања функције `f`. Потребна трансформација кода је сасвим једноставна, без обзира на потенцијалну сложеност функције `f`:

```
q()->map(\r:r$X)
   ->map(f)
where {
    q() = query {
        select X
        from ...
        };
    };
```

Пример 24: Израчунавање функције `f` на страни клијента

```
q()->map(\r:r$X)
where {
    q() = query {
        select :(f(X)) as X
        from ...
        };
    };
```

Пример 25: Израчунавање функције `f` на страни сервера

14.7.3. Управљање местом израчунавања

Када се представљена могућност премештања израчунавања на сервер упари са могућношћу дефинисања апстрактних полиморфних функција у програмском језику *Wafl*, стичу се услови да се управљање местом израчунавања премести из домена развоја у домен конфигурисања извршног система.

Пример 26 илуструје један начин имплементирања подсистема за одабирање места извршавања дате функције путем вредности конфигурационих параметара апликације. У конкретном примеру, параметар `location_f` означава да ли се израчунавање одвија на страни сервера или на страни клијента.

```
q()->map(\r:r$X)
   ->mapClientVersion(f)
where {
    q() = query {
        select :(serverVersion(f)(X)) as X
        from ...
        };

    serverVersion(f) =
        if ServiceValue['location_f'] = 'server'
            then f
            else \x:x;

    mapClientVersion(f) =
        if ServiceValue['location_f'] != 'server'
            then map(_,f)
            else \x:x;
    };
```

Пример 26: Пример одређивања места израчунавања на основу конфигурационог параметра `location_f`

14.7.4. Имплицитно дистрибуирање израчунавања

У случају система за управљање базама података који су подељени на више рачунарских система (тзв. *партиционисане базе података*), премештање сложених израчунавања на сервер представља истовремено и вид централизовања и вид дистрибуирања израчунавања:

- из аспекта апликације и развојног тима, израчунавање је централизовано на серверу и њиме се управља кроз упитне функције;
- из аспекта стварног места израчунавања, израчунавање је дистрибуирано на различите сервере који чине партиционисан СУБП, и њиме се управља имплицитно од стране СУБП-а.

Упаривањем описаног вида имплицитног дистрибуирања начелно централизованог израчунавања и представљене могућности управљања местом израчунавања кода кроз конфигурисање система стварају се предуслови за повећавање скалабилности система. Додатни квалитет је да се скалабилношћу може управљати кроз конфигурисање система, без потребе за значајним интервенцијама у коду.

14.8. Безбедносна питања

14.8.1. Потенцијални безбедносни проблеми

Неки од концепата представљени у овом раду могу да произведу одређене безбедносне проблеме. Уопштено посматрано, безбедносни пропусти могу да се траже на сваком месту на коме се приступа неким подацима или ресурсима било ради читања или писања. Различити програмски језици омогућавају рад са различитим врстама података и ресурса, што утиче и на разноликост могућих безбедносних проблема.

У случају програмског језика *Wafl*, програми могу да читају и мењају трајне податке искључиво путем система датотека и СУБП-а. Не постоји могућност непосредног приступања другим трајним ресурсима рачунарског система. Због тога се потенцијални безбедносни проблеми могу тражити у облику:

- неовлашћеног читања или мењања података путем система датотека;
- неовлашћеног читања или мењања података путем СУБП-а и
- ненамерног покретања злонамерног кода.

У случају програмског језика који омогућава непосредно приступање другим ресурсима система, могу постојати и друге врсте безбедносних проблема. Овде је од интереса првенствено анализа проблема који представљају последицу представљених концепата повезивања програмског језика са базом података. Други аспекти су углавном ортогонални у односу на ово повезивање. Штавише, безбедносна питања која се односе на извршавање програма на страни клијента се не разликују од општих безбедносних питања. Такође, ни околности разматрања проблема права приступања подацима базе података нису значајно измењене у односу на уобичајена програмска решења.

Специфичност описаних концепата повезивања је омогућавање извршавања матичних израза на серверима који управљају базама података. Ту је, пре свега, проблематична чињеница да се са правима извршавања кода на страни клијента и правима приступања бази података изводи покретање програмског кода матичних израза на страни сервера. При томе, матични изрази могу да користе услуге СУБП-а.

Матични изрази и база података

У случају *Wafl*-а, једини начин да се из матичног израза приступа некој бази података јесте путем управо оне исте већ успостављене сесије са базом у оквиру које је позвана кориснички дефинисана функција која извршава матични израз. Непосредна последица тога је да матични изрази имају идентична права у односу на базу података као и *Wafl* програми из којих се употребљавају упитни изрази у којима су ти матични изрази. Одатле непосредно следи да није могуће остварити неки нови вид неовлашћеног читања или мењања садржаја базе података помоћу матичних израза.

Ако би се под постојећим привилегијама могао злоупотребити матични израз у односу приступање садржају базе података, онда би се на потпуно исти начин могао злоупотребити и клијентски програм без матичних израза. Због тога се свако даље разматрање овог проблема своди на уобичајено разматрање проблема безбедности у базама података.

Са друге стране, ако би матични изрази могли да остварују везу са другим базама података, или под другачијим ауторизацијама него клијентски програми, онда би било веома важно сасвим недвосмислено одредити под којим условима и на који начин се тако нешто сме чинити. Све док се таква промена базе података или ауторизација изводи на исти начин као и на страни клијента, то не представља нов безбедносни проблем, јер се своди на случај приступања из клијентског програма. Због тога се може обликовати препорука да се *повезивање са базом података из матичних израза сме омогућити само на исти начин као из клијентског програма*.

Матични изрази и датотеке

Озбиљнији потенцијалан проблем је приступање систему датотека из матичних израза. Посебан проблем почива у чињеници да се матични израз израчунава у контексту процеса система за управљање базама података, а да тај процес уобичајено има изразито високе привилегије у односу на читав оперативни систем, па и на систем датотека. Последица тога је да постоји могућност да се путем матичних израза може приступати систему датотека са скоро неограниченим правима.

Ненамерно покретање злонамерног кода

Посебна врста проблема је евентуално ненамерно покретање злонамерног кода. Једна могућност је да се злонамерни код „увуче“ у текст (или превод) програма и самог матичног израза. Она се може предупредити добром заштитом програма и клијентских система. Друга могућност је да се при постављању злонамерног кода искористи могућност чувања програмског кода у табелама базе података. Тако постављен код би се могао прочитати и извршити на страни сервера.

Неусаглашеност верзија преводилаца

Потенцијалан извор проблема би могла да буде и евентуална некомпатибиност верзија преводилаца на страни клијента и сервера. У случају неусаглашености верзија

може се догодити да клијентски програм иницира извршавања неког матичног израза на серверу, који је савршено исправан у контексту верзије преводиоца која је на страни клијента, али који производи проблеме (нпр. грешке у току израчунавања или другачије понашање од очекиваног) на страни сервера.

Једино решење за ову врсту проблема је динамичко проверавање усаглашеноси верзија преводилаца.

14.8.2. Безбедносне мере

При тражењу могућих безбедносних мера се могу искористити могућности СУБП-а, ОС-а на серверу, ОС-а на клијенту, али и неке посебне могућности. Као што је већ указано, практично сви нови проблеми представљају последицу увођења концепта матичних израза. У складу са тиме, могуће безбедносне мере су:

- увођење дозвола за извршавање матичних израза;
- увођење дозвола за извршавање појединачних матичних израза и
- увођење дозвола за обављање одређених операција у матичним изразима.

Увођење општих дозвола за извршавања матичних израза

Матични изрази се извршавају посредством кориснички дефинисаних функција. Сваки СУБП располаже механизмима за управљање дозволама у односу на кориснички дефинисане функције. Корисник мора имати дозволу да користи неку функцију да би је могао извршити. То важи и за функције које израчунавају матичне изразе. У случају имплементације матичних израза помоћу кориснички дефинисаних функција, по узору на представљену експерименталну имплементацију, управљањем дозволама у односу на те функције се заправо остварује и управљање дозволама у односу на матичне изразе.

Ова врста дозвола се остварује на серверу база података и у односу на кориснички налог под којим се остварује комуникација програма на *Wafl*-у и базе података. Проблем са оваквим приступом је у ниском нивоу грануларности – корисник или може да израчунава све матичне изразе, или их уопште не може израчунавати.

Увођење дозвола за извршавање појединачних матичних израза

Управљање дозволама за извршавање појединачних израза се може остварити:

- на нивоу појединачних програма писаних на матичном језику;
- на нивоу појединачних матичних израза и
- на нивоу табела или колона у којима се записују матични изрази.

Сваки појединачан програм на матичном језику је записан у некој датотеци. Због тога је управљање дозволама на нивоу појединачних програма могуће остварити кроз механизме оперативног система, тј. система датотека на клијентском рачунару. Што се тога тиче, и проблем и решење су слични као у случају програма писаних на било ком програмском језику, независно од начина повезивања са базама података.

Управљање дозволама на нивоу појединачних матичних израза представља озбиљнији проблем. За његово решавање је неопходно обезбедити средство за једнозначно идентификовање матичних израза, као и механизме за прикупљање

информација о њима и управљање правима корисника. Међутим, колико год да се добро реши управљање матичним изразима на страни клијента, додатни проблем је управљање дозволама у односу на матичне изразе који се у читају и израчунавају на серверу. Најозбиљнији аспект проблема је могућност да се у оквиру једног матичног израза прочита и изврши неки други матични израз из базе података. Да би се могло управљати дозволама и у том домену, неопходно је увести додатне дозволе у односу на употребу неких могућности концепта матичних израза.

Решавање безбедносних проблема на овај начин би створило велики број нових проблема и значајно умањило употребљивост матичних израза, па се не препоручује.

Увођење дозвола за обављање одређених операција у матичним изразима

Неке аспекте безбедности је могуће решавати кроз увођење нових дозвола и механизама за њихово имплементирање. Штавише, то је и једини начин да се неки безбедносни проблеми реше без смањивања функционалности концепта матичних израза. У најосетљивије операције које се могу извршити у матичним изразима на серверу спада употреба датотека. Решење је могуће тражити у више смерова:

- потпуна забрана употребе датотека у матичним изразима;
- увођење дозвола за коришћење функција за рад са датотекама:
  - глобално, за све операције, кориснике, базе података и датотеке;
  - на нивоу операција: читање, писање, листање садржаја директоријума;
  - на нивоу база података;
  - на нивоу корисника;
  - на нивоу конкретних датотека или директоријума;
- коришћење датотека под дозволама корисника који је повезан са базом података, уместо под дозволама СУБП-а;
- омогућавање конфигурисања интерпретатора/преводиоца за матични језик на серверу тако да за сваку базу података могу да се дефинишу скупови датотека или директоријума који се могу користити из матичних израза.

Друга осетљива категорија операција се односи на приступање бази података из матичних израза и извршавање матичних израза похрањених у бази података. И у овом случају постоји више аспеката решења:

- потпуна забрана приступања бази података из матичних израза;
- увођење дозвола за приступање бази података из матичних израза:
  - глобално, за све кориснике и базе података;
  - појединачно, за конкретне кориснике и базе података;
  - ограничавање приступа табелама из матичних израза;
- увођење додатних дозвола за приступање атрибутима табела у којима се могу чувати изрази матичног језика:
  - за извршавање израза прочитаних из базе података;
  - за читање израза прочитаних из базе података.

Потпуна забрана приступања датотекама или бази података не би представљала добро решење, јер би водила значајном умањивању употребљивости матичних израза.

Типизираност и проблем безбедности

Строга типизираност повезивања програмског језика и базе података омогућава да се поуздано препознају атрибути базе података у којима се може налазити програмски код. На тај начин се може сузити скуп атрибута за које је потребно уводити додатне дозволе.

У случају евентуалне неусаглашености верзија преводилаца на страни клијента и сервера, мање је вероватно да проблематичан код буде успешно преведен на серверу уколико је језик строго типизиран. На тај начин се, у случају евентуалних проблема, они препознају у фази превођења матичног израза, а не у фази његовог извршавања. У случајевима када успешно извршавање матичног израза није могуће, оно се ни не започиње. Тиме се ублажавају последице неусаглашености верзија преводилаца.

14.9. Сагласност имплементације и концепата

Ради целовитости анализе особина предложеног решења и представљене имплементације неопходно је размотрити у којој мери је имплементација у складу са уведеним концептима. При описивању начина повезивања програмског језика *Wafl* са базама података, као и у оквиру представљања експерименталне имплементације, на више места је било речи о одређеним аспектима ове усаглашености. Табела 1 садржи упоредни преглед концепата и одговарајућих елемената програмског језика *Wafl.*

Концепт	Одговарајући елемент програмског језика *Wafl*
1. Употреба упитног језика у програмском језику	Упитне, акционе и трансакционе функције.
2. Употреба програмског језика у упитном језику	Матични изрази.
3. Строга типизираност повезивања	Подсистем за распознавање типова.
4. Јединствен трансакциони простор	Трансакционе функције. Скуп променљивих сесије. Акционе и упитне функције.
5. Ортогоналност у односу на типове података	Кориснички типови базе података. Табела пресликавања типова. Библиотеке кода у бази података.
6. Ортогоналсност у односу на начин повезивања	Основни начин рада је статички. Постоје функције које омогућавају динамички начин рада. Основни начин рада је строго типизирани. Постоји слабо типизирани начин рада. Динамички начин рада се одвија слабо типизирано. Основни начин читања је лењо читање. Вредно читање је подржано помоћу функције `forced`.

Табела 1: Концепти повезивања и елементи програмског језика *Wafl*

Употреба упитног језика у програмском језику је у потпуности испоштована спецификацијом упитних, акционих и трансакционих функција у *Wafl*-у. Што се тиче имплементације, ту остаје простор за унапређења. Манипулативни део *SQL*-а је подржан у потпуности, али део који се односи на дефинисање података није у потпуности подржан постојећим скупом управљачких модула.

И по питању употребе програмског језика у упитном језику, спецификација програмског језика *Wafl* је у потпусности у складу са концептима. Како је већ наглашено у одељку 14.1.1, имплементација има више ограничења. Није имплементирана подршка за колонске и табеларне матичне изразе, као ни пуна подршка за матичне изразе у упитним изразима унутар матичних израза.

Строга типизираност повезивања се остварује кроз сарадњу подсистема за распознавање типова преводиоца за програмски језик *Wafl* и одговарајућег подсистема за припрему упита СУБП-а. Усаглашавају се типови аргумената упитних функција и њихових резултата са остатком програма. Омогућена је и употреба слабо типизираних упита.

Трансакциони простор програмског језика *Wafl* обухвата базу података и променљиве апликативне сесије. Не обухвата систем датотека. Последица је да није остварена пуна јединственост трансакционог простора. Ипак, чак и оваква ограничена примена се у пракси добро показала. У неколико пројеката у којима је употребљаван програмски језик *Wafl* ([Miti2003], [Malk2008a], [Malk2008b] и други), у складу са специфичностима програмског језика акценат је стављен на употребу база података, док датотеке или уопште нису употребљаване или су употребљаване само за читање у сасвим ограниченом обиму. Концепт јединствених трансакција се показао одлично. Основна корист је управо она која је и очекивана – програмер не мора да се посебно брине о усклађивању промена података сесије са успехом или неуспехом одвијања трансакција, јер је старање о томе аутоматизовано у оквиру трансакције.

Дословно сви исправни типови у *Wafl*-у су подржани у бази података. Постоји простор за унапређивање дела имплементације који се односи на пресликавање типова програмског језика и базе података. Што се тиче типова базе података, требало би да се унапреди рад са временским типовима базе података у матичном језику, али ту одређени проблем представља и неуједначено имплементирање временских типова у СУБП-овима. Све то чини да је ортогоналност експерименталне имплементације у односу на типове података релативно добра.

Највећи проблем у односу на типове података представља чињеница да ни пуна примена распознавања типова података који се размењују између програма и базе података није довољна да се у потпуности избегну евентуални проблеми са неисправним типовима. До проблема може доћи ако се садржај базе података мења од стране других програма, писаних на другим програмским језицима, јер при томе може да буде заобиђено и проверавање исправности записа израза и проверавање усаглашености типова. Због тога је у имплементацију уграђена и додатна провера исправности типова сваког податка сложеног типа који се чита из базе података. Проблем са таквом имплементацијом је двојак – поред тога што се неминовно губи на перформансама, иако се на тај начин могу препознати неисправности, то се ипак дешава у фази извршавања програма, а не у фази превођења, како би било идеално. Овај проблем може да се реши

само доградњивањем СУБП-а механизмима за строго проверавање исправности података сложених типова.

У односу на све разматране различите начине повезивања и рада са подацима из базе података, представљена имплементација је ортогонална, у смислу да омогућава различите начине рада. Једино што није, нити може бити у потпуности подржано, јесте комбинација динамичког и строго типизираног начина рада. Један од начина превазилажења неких облика овог проблема је представљен у одељку 14.4. Други начин је да се на основу начина употребе уведу нека ограничења у одосу на типове резултата динамички израчунаваних упита. На пример, ако се резултат упита користи као да има неке атрибуте неког типа, онда се може претпоставити да резултат садржи бар те атрибуте. Проблем је у томе што провера сагласности стварног типа резултата са очекиваним типом може да се одвија само у фази извршавања програма.

14.10. Поређење са другим резултатима

Принципи и концепти изведени у овом раду се разликују у извесној мери од закључака Аткинсона и Бјунемана [Atki1987] (одељак 5.4). Највећа разлика је у томе што се у овој дисертацији уместо једног униформног језика предлаже употреба два различита језика који су међусобно повезани ортогонално и симетрично. Поред тога, принцип апстрактности налаже да повезивање мора да буде такво да се може на одговарајући начин имплементирати и на другим паровима упитних и програмских језика.

Ова разлика може деловати као веома значајна и непомирљива. Међутим, ако се претпостави да упитни језик може да ради искључиво са перзистентним подацима, а да програмски језик не може да изводи никакве бочне ефекте ни да непосредно приступа перзистентним подацима, онда се може рећи да се интеграцијом ових језика добија *један* нов униформан језик. Штавише, при разматрању концепта униформног језика потребно је узети у обзир циљеве који су таквим језиком требало да буду испуњени. Сасвим је очигледно да се ти циљеви, а пре свега Тјурингова комплетност, могу постићи ортогоналним и симетричним повезивањем програмског и упитног језика. Због тога је проблем који се у овом раду посматра општији од проблема који је посматран у [Atki1987].

Испоставља се да остали принципи из [Atki1987] могу да се изведу као последице принципа који су обликовани у овом раду. На пример, принципи (2), (3), (4) и (5) су непосредна последица принципа ортогоналности у односу на типове података. Сасвим је очигледно да их концепти представљени у овом раду у потпуности задовољавају. Принципи (8) и (9) су неминовна последица употребе строго типизираних програмских језика, па их у овом раду није било потребно експлицитно наводити. Такође, принципи (6) и (7) су нераскидиво повезани са природом конкретног програмских језика.

15. Закључак

Програмски језици и базе података постоје у различитим облицима практично од почетка развоја рачунарства. Због тога је већ дуго присутан и проблем повезивања програмских језика и база података, што има за резултат релативно велики број различитих решења тог проблема. У овом раду је начињен покушај систематизовања приступа проблему повезивања, са посебним акцентом на повезивање строго типизираних функционалних програмских језика и релационих база података.

У раду су анализирани различити аспекти повезивања програмских и упитних језика. Препозната су три основна принципа на којима мора почивати *добро* повезивање програмског и упитног језика, које не би доводило у питање постојеће карактеристике оба језика. То су:

- апстрактност,
- ортогоналност и
- симетричност.

Препознати су и концепти помоћу којих се може остварити повезивање у складу са претходним принципима. Концепти су детаљније разрађени за случај строго типизираног програмског језика, у виду обликовања одговарајућих потконцепата:

1. Употреба упитног језика у програмском језику:
   а. Упитне функције;
   б. Акционе функције;
2. Употреба програмског језика у упитном језику:
3. Строга типизираност повезивања:
   а. Омогућавање генеричког приступа подацима;
4. Јединствен трансакциони простор:
   а. Локализација бочних ефеката;
   б. Трансакционе функције;
   в. Акционе функције;
   г. Начин израчунавања упитних функција;
   д. Ниво изолованости трансакције;

ђ. Ниво изолованости упита;

е. Матични изрази;

5. Ортогоналност у односу на типове података:
   а. Програми у бази података;
   б. База података као библиотека кода;
6. Ортогоналност у односу на начин повезивања:
   а. Статички и динамички SQL;
   б. Строго и слабо типизирани резултати упита и
   в. Вредно и лењо читање.

Представљен је потпуно нов концепт матичних израза, који се израчунавају на страни СУБП-а при израчунавању упита. Представљена је и имплементација овог концепта за програмски језик *Wafl* и СУБП *IBM DB2*. Овај концепт се, по узору на представљену имплементацију, може имплементирати и за друге програмске језике и базе података. Његова примена је начелно могућа и у случају императивних програмских језика.

Концепт јединственог трансакционог простора није нов. Међутим, програмски језик *Wafl* је први функционалан програмски језик у коме се имплицитан механизам трансакција употребљава за синтаксно и семантичко заокруживање делова кода који остварују бочне ефекте или трпе последице остваривања бочних ефеката.

Представљен је један начин записивања података сложених типова у бази података у генеричком облику. Основна идеја није нова, али је овде примењена на нов начин, са алгоритмима који омогућавају проверавање типова тако похрањених података кроз сарадњу преводиоца за програмски језик и СУБП-а. Развијен је нов начин пресликавања система типова програмског језика у систем типова базе података, који почива на употреби постојећих концепата релационих база података.

Потврђена је хипотеза да се повезивање програмских језика и база података може заснивати на међусобној ортогоналности повезаних језика. Штавише, на конкретном примеру повезивања строго типизираног програмског језика *Wafl* и СУБП *IBM DB2* је потврђено да су понуђени концепти повезивања остварив уз пуно придржавање свих представљених основних принципа повезивања.

Размотрене су могућности примене представљеног интегрисаног решења у области дистрибуираног израчунавања. Развијена је и тестирана експериментална имплементација система за дистрибуирано израчунавање.

Добијени резултати могу да представљају основу за даља истраживања. Један смер истраживања би могао да се односи на одговарајуће систематизације повезивања програмских језика и база података са различитим врстама системских ресурса. Други могући предмет истраживања, у великој мери повезан са претходним, би могла да буде даља разрада концепта јединственог трансакционог простора, уз обухватање различитих врста ресурса рачунарског система.

У овом раду је указано на проблем проширивања система типова база података тако да обухвате и општије системе типова програмских језика. Указано је да, због високог нивоа спрегнутости тог проблема са конкретним имплементацијама база података, такав

приступ није погодан у претходној фази истраживања. Међутим, сада на основу добијених резултата оствареног истраживања има смисла ићи и корак даље у том смеру.

Било би веома интересантно размотрити могућности имплементирања концепта матичних израза у императивним програмским језицима. Таква имплементација би вероватно морала да буде значајно сложенија него што је то случај када су у питању функционални језици.

Могући смер истраживања представља и примена представљеног решења у области аутоматизације извођења закључака о програмима, укључујући доказивање исправности програма и оптимизацију. При анализирању сложенијих програма се производе велике количине података, чијем обрађивању би могли да допринесу представљени концепти употребе база података.

Део V

Додаци

Додатак А - Конфигурисање повезивања *Wafl* програма са базом података

Успостављање комуникације између програма писаног на програмском језику *Wafl* и СУБП-а се одвија аутоматски, на основу вредности одговарајућих конфигурационих параметара апликације. Вредности конфигурационих параметара одређује администратор Веб апликације у посебној конфигурационој датотеци. Табела 2 садржи преглед параметара који се односе на повезивање са базом података. Поред тих параметара, конфигурациона датотека садржи већи број параметара који се односе на начин рада Веб апликације [Малк2002], а који у овом контексту нису посебно значајни.

Параметар	Значење
`dbDriver`	Ниска која идентификује управљачки модул за конкретан СУБП. Параметар није обавезан. Подразумевана вредност је `ODBC`. Може бити `DB2v9`, `DB2v8` и друго.
`dbAlias`	Ниска која једнозначно идентификује базу података коју апликација употребљава. Облик и садржај ниске могу зависити од имплементације СУБП-а, окружења и изабраног управљачког програма.
`dbUser`	Ниска која представља корисничко име под којим програм употребљава базу података.
`dbPassword`	Ниска која представља лозинку уз наведено корисничко име.
`dbKeepSessions`	Највећи допуштен број неактивних веза са базом података које се чувају ради уштеде ресурса при поновном успостављању везе.
`dbQueryIsolation`	Подразумевани ниво изолованости упита (за случај постављања упита ван експлицитно дефинисаних трансакција). Допуштене вредности су `s`, `rr`, `rc`, `ru`. Параметар није обавезан. Подразумевана вредност је `rc`.
`dbTransIsolation`	Подразумевани ниво изолованости трансакција. Допуштене вредности су `s`, `rr`, `rc`, `ru`. Параметар није обавезан. Подразумевана вредност је `rr`.

Табела 2: Параметри апликације који одређују начин рада са базом података

Додатак Б - Табеле које се употребљавају у примерима

У примерима у овом раду су употребљаване табеле упрошћеног модела базе података факултета. Овде је приложен опис употребљаваних табела, без намере да се детаљно излаже њихова семантика.

Табела `student` садржи податке о студентима који су уписани на факултет. Атрибут `indeks` је цео број облика `godUpisa*10000+brojUGodini`:

```
create table wafldb.student (
    indeks          integer not null,

    ime             varchar(50) not null,
    prezime         varchar(50) not null,

    primary key( indeks )
);
```

Табела `predmet` садржи податке о предметима који постоје на факултету:

```
create table wafldb.predmet (
    id              integer not null,

    sifra           varchar(20) not null,
    naziv           varchar(200) not null,
    semslus         smallint not null
        constraint chk_semslus
        check( semslus between 1 and 2),
    espb            smallint not null
        constraint chk_espb
        check( espb between 1 and 120),

    primary key( id ),

    constraint uk_sifra unique ( sifra )
);
```

Табела `ispit` садржи податке о положеним испитима:

```
create table wafldb.ispit (
    indeks              integer not null,
    id_predmeta         integer not null,
```

```
    ocena               smallint not null
        constraint chk_ocena
        check( ocena between 6 and 10 ),
    nastavnik           varchar(200) not null,
    dat_polaganja       date not null,
    napomena            varchar(2000),

    primary key( indeks, id_predmeta ),

    foreign key fk_pp_student( indeks )
        references wafldb.student,

    foreign key fk_pp_predmet( id_predmeta )
        references wafldb.predmet
);
```

У неким примерима, који се односе на записивање података сложених типова у бази података, употребљавају се табела `intlists` и одговарајући тип `list_int`:

```
create distinct type list_int as long varchar;

insert into wafl.dtypes
values ('list_int', 'List[Integer]');

create table wafldb.intlists(
    id  integer not null,
    list    list_int not null,
    primary key( id )
);
```

Додатак B - Функције за израчунавање матичних израза за СУБП *DB2*

15.1.1. Програм createWaflEvalUdfCpp.wafl

Програм `createWaflEvalUdfCpp.wafl` прави следеће датотеке, које се затим употребљавају при превођењу библиотеке `waflevaludf.cpp`:

- `generatedfunctions.h` – функције којима се имплементира већи број међусобно сличних функција за израчунавање *Wafl* израза;
- `waflevaludf.def` – декларација функција које улазе у динамичку библиотеку (*DLL*);
- `Create.Functions.sql` – скрипт наредби *SQL*-а којима се праве одговарајуће функције у бази података.

```
//-------------------------------------------------------
//  createWaflEvalUdfCpp.wafl
//  =========================
//  Програм који прави програмске датотеке које имплементирају
//  кориснички дефинисане функције (UDF):
//    - generatedfunctions.h
//          имплементација на C-у, која се укључује у модул waflevaludf.cpp
//    - waflevaludf.def
//          дефиниција динамичке библиотеке са UDF
//    - Create.Functions.sql
//          SQL скрипт за прављење UDF
//-------------------------------------------------------
makeAll(16)

//-------------------------------------------------------
where{

makeAll(n) =
    makeCode(n)->saveToFile("generatedfunctions.h")
    and makeSql(n)->saveToFile("Create.Functions.sql")
    and makeDef(n)->saveToFile("waflevaludf.def")
    ;

makeCode(n) =
    '/*\r\n'
```

```
        '    DO NOT CHANGE!\r\n'
        '    AUTOMATICALLY GENERATED FILE! \r\n'
        '*/\r\n\r\n\r\n'
        + makeAndJoinCodeSegmentsList( n,
            [function_wafl_evaluate, function_wafl_evaluate_dt],
            '\r\n\r\n\r\n', '\r\n\r\n\r\n'
            )
        ;

    makeSql(n) =
        '\r\n'
        '-- DO NOT CHANGE! \r\n'
        '-- AUTOMATICALLY GENERATED FILE!\r\n\r\n\r\n'
        + makeAndJoinCodeSegmentsList( n,
            [   sql_drop_wafl_function,
                sql_create_wafl_function,
                sql_drop_wafl_function_dt,
                sql_create_wafl_function_dt ],
            '\r\n', '\r\n\r\n\r\n'
            )
        ;

    makeDef(n) =
        '\r\n'
        '; DO NOT CHANGE! \r\n'
        '; AUTOMATICALLY GENERATED FILE!\r\n\r\n\r\n'
        'LIBRARY WAFLEVALUDF\r\n'
        'EXPORTS\r\n'
        + makeList(0,n)
            ->map(\n: '    wafl_evaluate_' + n->asString() + '\r\n'
                      '    wafl_evaluate_dt_' + n->asString() )
            ->strJoin('\r\n')
        ;

    //-------------------------------------------------------
    makeAndJoinCodeSegmentsList(n,l,sep1,sep2) =
        l->map( makeAndJoinCodeSegments(n,_,sep1) )
         ->strJoin(sep2)
        ;

    makeAndJoinCodeSegments(n,fn,sep) =
        makeList(0,n)
            -> map(fn)
            ->strJoin(sep)
        ;

    makeList(n,m) =
        if n > m then []
        else n : makeList(n+1,m)
        ;

    saveToFile(c,fn) =
        fileWrite( fn, c )
            ->strLen()
            > 0
        ;

    repeat(s,n) =
        if n<1 then ''
        else s + repeat(s,n-1);

    //-------------------------------------------------------
    sql_drop_wafl_function(n) =
        'drop function wafl.WaflEvaluate( LONG VARCHAR'
            + repeat(', LONG VARCHAR', n )
            + ' );'
        ;

    //-------------------------------------------------------
    sql_create_wafl_function(n) = html template
    Create function wafl.WaflEvaluate( LONG VARCHAR<#
            repeat(', LONG VARCHAR', n ) #> )
```

```
    returns LONG VARCHAR
    external name 'waflevaludf.dll!wafl_evaluate_<# n->asString() #>'
    language c
    parameter style db2sql
    not deterministic
    reads sql data
    external action
    fenced;

Grant Execute on
    function wafl.WaflEvaluate( LONG VARCHAR<# repeat(', LONG VARCHAR', n) #> )
    to public;
<#>;

//-------------------------------------------------------
sql_drop_wafl_function_dt(n) =
    'drop function wafl.WaflTypedEvaluate( LONG VARCHAR'
        + repeat(', LONG VARCHAR, INTEGER', n )
        + ' );'
    ;

//-------------------------------------------------------
sql_create_wafl_function_dt(n) = html template
Create function wafl.WaflTypedEvaluate( LONG VARCHAR<#
        repeat(', LONG VARCHAR, INTEGER', n ) #> )
    returns LONG VARCHAR
    external name 'waflevaludf.dll!wafl_evaluate_dt_<# n->asString() #>'
    language c
    parameter style db2sql
    not deterministic
    reads sql data
    external action
    fenced;

Grant Execute on
    function wafl.WaflTypedEvaluate( LONG VARCHAR<#
        repeat(', LONG VARCHAR, INTEGER', n ) #> )
    to public;
<#>;

//-------------------------------------------------------
function_wafl_evaluate(n) = html template

/*************************************************************
* wafl_evaluate_<# n->asString() #>
*   функција која израчунава Wafl израз са <# n->asString() #> аргумената
*     улазни:  LONG VARCHAR  code <#
*           if n>0 then
*               '\r\n*     input:  LONG VARCHAR  arg1..arg'
*               + n->asString()
*           else ''
*       #>
*     излазни: LONG VARCHAR  out
*************************************************************/
SQL_API_RC SQL_API_FN wafl_evaluate_<# n->asString() #>(
    SQLUDF_LONG     *code,      /* улазни:  текст програма */<#
        makeList(1,n)
            ->map(\n: '\r\n    SQLUDF_LONG         *arg'
                      + n->asString()
                      + ',      /* улазни:  аргумент */')
            ->strJoin('')
        #>
    SQLUDF_LONG     *out,       /* излазни: резултат као ниска */
    SQLUDF_NULLIND  *codenull,  /* NULL индикатор */<#
        makeList(1,n)
            ->map(\n: '\r\n    SQLUDF_NULLIND      *arg'
                      + n->asString()
                      + 'null,      /* NULL индикатор */')
            ->strJoin('')
        #>
    SQLUDF_NULLIND  *outnull,   /* NULL индикатор */
    SQLUDF_TRAIL_ARGS)          /* пратећи аргументи */
```

```
{
    if( *codenull <#
        makeList(1,n)
            ->map(\n:'|| *arg' + n->asString() + 'null ')
            ->strJoin('')
    #> ){
        *outnull = -1;
        return 0;
        }

    PREPARE;<#
        makeList(1,n)
            ->map(\n: '\r\n    ADDGENERICARG(' + n->asString() + ');')
            ->strJoin('')
    #>
    return processStringResult(req,out);
}
<#>;

//-------------------------------------------------------
function_wafl_evaluate_dt(n) = html template

/*************************************************************
*  wafl_evaluate_dt_<# n->asString() #>
*    функција која израчунава Wafl израз са <#
*       n->asString() #> типизираних аргумената
*     улазни:  LONG VARCHAR  code <#
*       if n>0 then
*           '\r\n*     улаз:  LONG VARCHAR  arg1..arg'
*           + n->asString()
*       else ''
*   #><#
*       if n>0 then
*           '\r\n*     улаз:  INTEGER  arg1type..arg'
*           + n->asString() + 'type'
*       else ''
*   #>
*     излазни: LONG VARCHAR  out
*************************************************************/
SQL_API_RC SQL_API_FN wafl_evaluate_dt_<# n->asString() #>(
    SQLUDF_LONG         *code,      /* input:  program text */<#
        makeList(1,n)
            ->map(\n:
                '\r\n    SQLUDF_LONG         *arg'
                + n->asString()
                + ',           /* улазни:  аргумент */'
                '\r\n    SQLUDF_INTEGER      *arg'
                + n->asString()
                + 'type,       /* улазни:  тип аргумента */'
                )
            ->strJoin('')
        #>
    SQLUDF_LONG         *out,       /* излазни: резултат као ниска */
    SQLUDF_NULLIND      *codenull,  /* NULL индикатор */<#
        makeList(1,n)
            ->map(\n:
                '\r\n    SQLUDF_NULLIND      *arg'
                + n->asString()
                + 'null,       /* NULL индикатор */'
                '\r\n    SQLUDF_NULLIND      *arg'
                + n->asString()
                + 'typenull,  /* NULL индикатор */'
                )
            ->strJoin('')
        #>
    SQLUDF_NULLIND      *outnull,   /* NULL индикатор */
    SQLUDF_TRAIL_ARGS)              /* пратећи аргументи */
{
    if( *codenull <#
        makeList(1,n)
            ->map(\n: '|| *arg' + n->asString() + 'null '
                      '|| *arg' + n->asString() + 'typenull ' )
```

```
            ->strJoin('')
        #> ){
        *outnull = -1;
        return 0;
        }

    PREPARE;<#
        makeList(1,n)
            ->map(\n: '\r\n    ADDTYPEDARG(' + n->asString() + ');')
            ->strJoin('')
    #>
    return processStringResult(req,out);
}

<#>;

//-------------------------------------------------------
}
```

Програм 4: Аутоматско прављење програмског кода за кориснички дефинисане функције за израчунавање матичних израза

15.1.2. Програм waflevaludf.cpp

Програм `waflEvalUdf.cpp` садржи главне делове функција за израчунавање *Wafl* израза. Саме функције се укључују са датотеком `generatedfunctions.h`. Написан је на програмском језику *C++* уз одређене специфичности које се односе на оперативни систем *Windows*.

```
#include <windows.h>
#include <stdio.h>
#include <sqlsystm.h>
#include <sqludf.h>
#include <string.h>

#include "../appHandlerDll/HandlerInterface.h"

//------------------------------------------------------------------
//  Главна функција модула
BOOL WINAPI DllMain(
    HINSTANCE hinstDLL,  // handle to DLL module
    DWORD fdwReason,     // reason for calling function
    LPVOID lpReserved )  // reserved
{
    // У зависности од разлога позивања...
    switch( fdwReason )
    {
        case DLL_PROCESS_ATTACH:
            // Једнократна иницијализација за сваки нов процес.
            // Враћа FALSE ако не успе читање библиотеке.
            if( LoadWaflHandlerDll(
                    &waflHandler,
                    "..\\function\\Handler.dll" ))
                return FALSE;
            break;

        case DLL_PROCESS_DETACH:
            // Потребна ослобађања ресурса.
            FreeWaflHandlerDll(&waflHandler);
            break;
    }
    return TRUE;  // Успешно.
}

//------------------------------------------------------------------
//  Помоћне функције
char* terminatedLongVarchar( SQLUDF_LONG *s )
```

```
{
    if( s->length < 32700 )
        s->data[s->length] = 0;
    else
        s->data[32700] = 0;
    return s->data;
}

void setLibError( SQLUDF_LONG *out )
{
    strcpy(
        out->data,
        waflHandler.loadingErrorCode==1
            ? "Can not load Wafl dynamic library!"
        : waflHandler.loadingErrorCode==2
            ? "Can not find necessary functions in Wafl library!"
            : "Unknown error!"
        );
    out->length = strlen( out->data );
}

//-------------------------------------------------------------
//  Делови поступка, описани у облику макроа
#define PREPARE                                                 \
    *outnull = 0;                                               \
    if(!waflHandler.hLib){                                      \
        setLibError(out);                                       \
        return 1;                                               \
        }                                                       \
    WaflHandlerRequest* req = waflHandler.AllocateRequest();    \
    req->ArgCount = 0;                                          \
        req->ProgramCode = terminatedLongVarchar( code )

#define ADDGENERICARG(n)                                \
    req->ArgumentType[req->ArgCount] = WDT_GENERIC;     \
    req->ArgumentLen[req->ArgCount] = arg##n->length;   \
    req->Argument[req->ArgCount++] =                    \
        terminatedLongVarchar( arg##n )

void addTypedArg(
        WaflHandlerRequest* req,
        SQLUDF_LONG* arg,
        SQLUDF_INTEGER* argtype
        )
{
    int i = req->ArgCount;
    req->ArgumentType[i] = *argtype;

    switch( *argtype ){
        case WDT_INTEGER:
            sscanf( terminatedLongVarchar( arg ),
                    "%d", (int*)&req->Argument[i] );
            break;
        case WDT_BOOL:
            *(int*)&req->Argument[i] =
                *arg->data!='f'
                && *arg->data!='F'
                && *arg->data!='0';
            break;
        case WDT_FLOAT:
            sscanf( terminatedLongVarchar( arg ),
                    "%f", (float*)&req->Argument[i] );
            break;
        case WDT_STRING:
        case WDT_GENERIC:
            req->ArgumentLen[i] = arg->length;
            req->Argument[i] = terminatedLongVarchar( arg );
            break;
        default:
            req->ArgumentLen[i] = 0;
            req->Argument[i] = "";
        }
```

```
    req->ArgCount++;
}

#define ADDTYPEDARG(n)                          \
    addTypedArg( req, arg##n, arg##n##type )

int processStringResult(WaflHandlerRequest* req, SQLUDF_LONG* out)
{
    int succ = waflHandler.HandleRequest( req );
    switch( req->ResultType ){
        case WDT_INTEGER:
            sprintf( out->data, "%ld", *(int*)&req->Result );
            out->length = strlen( out->data );
            break;
        case WDT_BOOL:
            out->data[0] = *(int*)&req->Result ? 'T' : 'F';
            out->length = 1;
            break;
        case WDT_FLOAT:
            sprintf( out->data, "%f", *(float*)&req->Result );
            out->length = strlen( out->data );
            break;
        case 0:
        case WDT_STRING:
        default:
            out->length = min(req->ResultLen,32700);
            memcpy( out->data, req->Result, out->length );
            break;
        }
    waflHandler.FreeRequest( req );
    return 0;
}

//-------------------------------------------------------------
//  Укључивање програмски направљених делова кода...
#include "generatedfunctions.h"
```

Програм 5: Главни модул библиотеке корисничких функција за израчунавање матичних израза

Додатак Г - Пример имплементације подсистема за дистрибуирано израчунавање

База података

Пример је обликован тако да омогућава строго типизирано дистрибуирање израчунавања послова који за резултат имају цео број. Ради подршке је уведен кориснички тип података `fun_int` који представља функцију без аргумената са целобројним резултатом:

```
create distinct type fun_int as long varchar;

insert into wafl.dtypes
values ('fun_int', '(->Integer)');
```

Пример 27: Типови за подршку дистрибуираног израчунавања

Подаци о пословима се записују у табели `wafljob.intjob`. Сваки посао има идентификатор (`ID`) и програмски код посла (`DESC`). При прављењу посла записује се време прављења (`CREATED`). При распоређивању посла се записује јединствена идентификација процеса (`PROCESSID`) и време започињања израчунавања (`STARTED`). По завршетку израчунавања се записују резултат (`RESULT`) и време завршавања (`FINISHED`).

Додатни податак о последњем додељеном идентификатору посла се води у табели `wafljob.state`.

```
create table wafljob.job(
    id          integer not null,   -- јединствени идентификатор посла
    desc        fun_int not null,   -- посао
    created     timestamp not null, -- време прављења
    started     timestamp,          -- време започињања израчунавања
    finished    timestamp,          -- време довршавања посла
    processid   varchar(100),       -- процес који израчунава посао
    result      integer,            -- резултат израчунавања
    primary key(id)
);

create index wafljob.intjob_started
    on wafljob.intjob(started,id);
```

```
create table wafljob.state(
    lastid      integer not null with default 0,
    primary key(lastid)
);

insert into wafljob.state values (0);
```

Пример 28: Табеле за чување података о пословима који се дистрибуирано израчунавају

Програм dist.wafl

Програм `dist.wafl` дистрибуира израчунавање 200 задатака. Задаци који се израчунавају су једноставни, али процесорски захтевни.

```
dist()

where   {
    dist() =
        mklist(3501,3700)
        ->distLib::distMap( ssum )
        ->map(\r:r$result)
        ;

    ssum(n) = if n<1 then 0 else sum(n) + ssum(n-1);
    sum(n) = if n<1 then 0 else n + sum(n-1);

    mklist(n,m) = if n>m then []
                          else n : mklist(n+1,m);

    distLib = library file 'distributed.wlib';
}
```

Пример 29: Програм који дистрибуира неке делове израчунавања

Програм run.wafl

Програм `run.wafl` је агент који израчунава направљене задатке. Претпоставља се да истовремено ради више инстанци овог програма. Значајан део програма `run.wafl` се односи на исписивање контролних порука на конзоли, како би се могло пратити израчунавање. Програм израчунава послове све док не буде прекинут.

```
run()

where   {
    run() = loop( processSingleJob );

    loop(f) = f()->(\x#f:loop(f))();

    processSingleJob() = distLib::waitForOpenJob()
                            ->map(process);

    process(id) = reportStart(id)
                    ->distLib::processJob()
                    ->reportEnd();

    reportStart( id ) =
        id->distLib::echo( distLib::mkTmpStr(
                            'Processing job ' + id->asString() + ' ...' ));

    reportEnd( jobResult ) =
        cmdPrint(
            jobResult$id->asString()
            + '  (' + jobResult$status->asString() + ')'
            + '  -->  ' + jobResult$result->asString()
```

```
            + '\n');

    distLib = library file 'distributed.wlib';
}
```

Пример 30: Агент дистрибуираног израчунавања

Библиотека distributed.wlib

Библиотека `distributed.wlib` садржи функције које се употребљавају при постављању и израчунавању послова.

```
library DistributedLib {

    // Додавање новог посла.
    //  - Аргумент је изворни код посла.
    //  - Резултат је идентификатор посла.
    createJob( fun ) = addJobWithId( dbGetNextId(), fun );

    // Израчунавање листе отворених послова.
    //  - Аргумент је максималан број послова који се издвајају.
    getOpenJobs(n) = q()
            ->(if n>0 then subList(_,0,n) else \x:x)()
            ->map(\r:r$ID)
    where{
        q() = typed sql query {
            select id
            from wafljob.intjob
            where started is null
            order by rand()
            };
        };

    // Чекање на отворен посао.
    //  - Резултат је листа ID посла или празна листа.
    waitForOpenJob() = wait(100000)
    where{
        wait(n) = getOpenJobs(1)
                    ->(\x,n:
                        if not x->empty()
                            then x
                        else if metaWaitUntil(100,100,\:false) or n<1
                            then []
                            else wait(n-1)
                    )(n);
        };

    // Чекање на резултат израчунатог посла.
    //  - Аргумент је ID посла.
    //  - Резултат је облика {status:; result:}
    //      status је
    //          N – ако посао не постоји
    //          P – ако посао чека (pending)
    //          S – ако је посао започет (started)
    //          F – ако је посао довршен (finished)
    waitJobResult(id) = waitN(id,1000)
                        ->(\s,id:
                            if s$status='F'
                                then s
                                else waitJobResult(id)
                        )(id)
    where{
        waitN(id,n) = dbJobStatusAndResult(id)
                        ->(\s,id,n:
                            if s$status='F'
                                then s
                            else if !metaWaitUntil(50,100,\:false) and n<=0
                                then s
                            else
```

```
                                waitN(id,n-1)
                        )(id,n)
                        ;
        };

    // Дистрибуирање послова и цекање на њихов резултат
    // Argumenti su:
    //  - листа података
    //  - функција која се примењује на елементе листе
    distMap( l, f ) =
        l->map( doCreateJob(f,_) )
         ->map( waitJobResult )
    where {
        doCreateJob( f, x ) = (\#f,x:f(x))->createJob();
        };

    // Израчунавање једног евидентираног посла.
    //  - Аргумент је ID посла.
    //  - Резултат је облика {id,status,result}
    //      status = true - акко је све у реду
    //      result = резултат или опис грешке, ако је status=false
    processJob( id ) =
        if registerStart( id, getProcessId() )
            then process( id, dbGetJobSource( id ) )
            else { id: id,
                   status:false,
                   result:0,
                   msg:'already locked' }
    where {
        process( id, job ) = id->save( job() );

        save( id, result ) = if dbSaveResult( id, result )
                             then { id: id,
                                    status: true,
                                    result: result,
                                    msg: 'OK' }
                             else { id: id,
                                    status: false,
                                    result: result,
                                    msg: 'Error' };

        getProcessId() = answerAction();

        registerStart( id, processid ) = sql transaction {
            update wafljob.intjob
            set started = current timestamp,
                processid = :processid
            where started is null
              and id = :id;

            not qSelected(id,processid).empty();
            };

        qSelected(id,processid) = typed sql query {
            select id
            from wafljob.intjob
            where id = :id
              and processid = :processid
            };
        };

    //  Помоћне функције за извештаје
    echo(x,s) = if cmdPrint(s)->isNull() then x else x;
    strRep(s,n) = if n<=0 then '' else strRep(s,n-1) + s;
    mkTmpStr(s) = s + strRep('\10',s->strLen());
}

where   {
    // Израчунавање (и резервисање) првог слободног id-а посла
    dbGetNextId() = if tIncreaseIdSequence()
                    then SessionValue('nextId').asInt()
                    else dbGetNextId()
```

```
        where {
            tIncreaseIdSequence() = sql transaction {
                update wafljob.state
                set lastid = lastid + 1;

                set nextId = qLastId().hd()$LASTID;
                };

            qLastId() = typed sql query {
                select lastid from wafljob.state
                };
            };

        // Додавање новог посла
        addJobWithId( id, src )= if dbAddJob( id, src )
                                    then id
                                    else 0;

        dbAddJob( id, fun ) = sql transaction {
            insert into wafljob.intjob( id, desc )
            values (:id, :fun);
            };

        // Читање посла.
        dbGetJobSource(id) = qGetJobSource(id)
                               ->map(\r:r$DESC)
                               ->(\l: if l->empty() then \:0 else l->hd() )()
        where{
            qGetJobSource(id) = typed sql query{
                select *
                from wafljob.intjob
                where id = :id
                };
            };

        // Записивање израчунатог резултата.
        dbSaveResult( id, result ) = sql transaction {
            update wafljob.intjob
            set result = :result,
                finished = current timestamp
            where id = :id
              and result is null;
            };

        // Читање резултата посла и провера да ли је довршен.
        //  - Аргумент је ID посла.
        //  - Резултат је резултат израчунатог посла,
        //    или празна ниска ако није израчунат.
        dbJobStatusAndResult(id) =
            q(id)
                ->map(\r: {status: r$STATUS, result:r$RESULT})
                ->(\l:
                    if l->empty()
                        then {status:'N', result:0}
                        else l->hd()
                    )()
        where{
            q(id) = typed sql query {
                select case
                        when finished is not null then 'F'
                        when started is not null then 'S'
                        else 'P'
                    end as status,
                    result
                from wafljob.intjob
                where id = :id
                };
            };
    }
```

Пример 31: Библиотека функција за дистрибуирано израчунавање

Напомене о имплементацији

Пример је написан тако да омогућава дистрибуирање израчунавања послова који за резултат имају цео број. У потпуности је строго типизиран. Уз минималне измене се може прилагодити неком другом типу послова.

Имплементација се једноставно може прилагодити и раду са произвољним типовима послова, али тада више не би била строго типизирана. Ради прилагођавања произвољном типу послова потребно је:

- променити типове колона које се односе на запис посла и резултата посла тако да обе подржавају рад са нискама;
- за израчунавање канонизованог програмског кода посла је потребно експлицитно употребљавати функцију `metaGetSource(exp)` и
- за израчунавање вредности посла (датог у текстуалном облику) је потребно употребљавати функцију `metaEvaluate(src)`.

Тако остварена имплементација сама по себи није строго типизирана, иако свака њена исправна употреба јесте строго типизирана. Мотивација за такву имплементацију долази из потребе да библиотека ради за све различите случајеве дистрибуирања послова.

У имплементацији се употребљава и функција `metaWaitUntil`. Она служи за синхронизацију и омогућава програму да пређе у неактивно стање. Има тип `(Integer * Integer * (-> Bool) -> Bool)`. Употребљава се у облику:

```
metaWaitUntil(period,limit,cond)
```

где је `period` цео број који описује после колико милисекунди се проверава испуњеност услова `cond`; `limit` је највеће допуштено укупно време чекања, ако се прекорачи резултат је `false`; `cond` је функција без аргумената која проверава испуњеност услова који се чека – први пут када услов буде испуњен завршава се израчунавање и резултат функције `metaWaitUntil` је `true`.

Резултати тестирања библиотеке

Модел је тестиран дистрибуирањем задатака различите сложености у окружењу са неколико рачунара истих карактеристика. Посматрано је колики су ефекти при дистрибуираном израчунавању. Табела 3 садржи добијене податке.

Показује се да на ефикасност значајно утиче сложеност послова који се дистрибуирају. То је, пре свега, последица увођења рада са базом података у послове који би иначе радили независно од базе података. Сваки посао се мора уписати у базу података, прочитати из ње, означити као распоређен, а на крају се и резултат израчунавања уписује и чита из базе података. У условима тестирања рад са базом података је доводио до просечног успорења од 0,25s по дистрибуираном послу. У случају већих послова се мање осећа успоравање које уводи рад са базом података и ефикасност израчунавања је далеко већа.

Добијени резултати су у границама очекивања и потврђују претпоставку да се представљен концепт повезивања функционалног програмског језика и релационе базе података може употребити за имплементацију система за дистрибуирано израчунавање.

Посебан квалитет представља једноставност програмског кода којим је обезбеђено дистрибуирано израчунавање, јер пружа широк простор за даља унапређења, којима се може добити и на перформансама и на могућностима.

Сложеност посла	Број послова	Број система	Укупно трајање	Коеф. убрзавања	Коеф. ефикасности
0,067s	1000	-[48]	67s	1	1
0,067s	1000	1	308s	0,22	0,22
0,067s	1000	2	217s	0,31	0,16
0,067s	1000	4	158s	0,42	0,11
2,6s	200	-	522s	1	1
2,6s	200	1	573s	0,91	0,91
2,6s	200	2	291s	1,79	0,90
2,6s	200	4	168s	3,11	0,78

Табела 3: Резултати тестирања ефикасности библиотеке за дистрибуирано израчунавање

[48] Случај израчунавања без дистибуирања је означен цртицом.

Додатак Д - Пример примене алгоритма распознавања типова

Нека је дат пример кода:

```
...
fact(n) =
    if n>1
        then n * fact(n-1)
        else 1;
...
```

Прављење скупа хипотеза

Први корак представља прављење скупа хипотеза, редом:

- на основу правила дефиниције за функцију `fact`:

  Tip(`fact`) = ’1 -> ’2 (x.1)
  Tip(izraz 1: `if`...) = ’2 (x.2)

- на основу правила сложеног израза за израз из хипотезе 2, при чему се инстанцирањем полиморфног типа оператора `if` од типа Bool * ’1 * ’1 -> ’1 добија тип Bool * ’3 * ’3 -> ’3:

  Tip(izraz 1: `if`...) = TipRes(`if`) = ’3 (x.3)
  Tip(izraz 2: `n>1`) = Bool (x.4)
  Tip(izraz 3: `n * fact(n-1)`) = ’3 (x.5)
  Tip(izraz 4: `1`) = ’3 (x.6)

- на основу правила сложеног израза за израз из хипотезе 4, при чему се инстанцирањем полиморфног типа оператора > од типа Value[’1] * Value[’1] -> Bool добија тип Value[’4] * Value[’4] -> Bool:

  Tip (izraz 2: `n>1`) = TipRes(>) = Bool (x.7)
  Tip(izraz 5: `n`) = Value[’4] (x.8)
  Tip(izraz 6: `1`) = Value[’4] (x.9)

- на основу правила простог израза за израз из хипотезе 8, где је `n` аргумент функције:

$$\text{Tip(izraz 5: n)} = \text{'1} \qquad \text{(x.10)}$$

- на основу правила простог израза за израз из хипотезе 9, где је `1` константа типа Integer:

$$\text{Tip(izraz 6: 1)} = \text{Integer} \qquad \text{(x.11)}$$

- на основу правила сложеног израза за израз из хипотезе 5, при чему је инстанцирањем полиморфног типа оператора `*` од типа:
  Numeric['1] * Numeric['1] -> Numeric['1]
  добијен тип:
  Numeric['5] * Numeric['5] -> Numeric['5]:

$$\text{Tip(izraz 3: n * fact(n-1))} = \text{TipRes(*)} = \text{Numeric['5]} \qquad \text{(x.12)}$$
$$\text{Tip(izraz 7: n)} = \text{Numeric['5]} \qquad \text{(x.13)}$$
$$\text{Tip(izraz 8: fact(n-1))} = \text{Numeric['5]} \qquad \text{(x.14)}$$

- на основу правила простог израза за израз из хипотезе 13, где је `n` аргумент функције:

$$\text{Tip(izraz 7: n)} = \text{'1} \qquad \text{(x.15)}$$

- на основу правила сложеног израза на израз из хипотезе 14, где је за функцију `fact` претпостављен тип: `'1 -> '2`:

$$\text{Tip(izraz 8: fact(n-1))} = \text{TipRes(fact)} = \text{'2} \qquad \text{(x.16)}$$
$$\text{Tip(izraz 9: n-1)} = \text{'1} \qquad \text{(x.17)}$$

- на основу правила сложеног израза на израз из хипотезе 17, при чему је полиморфни тип оператора `-` инстанциран, тако да је од типа
  Numeric['1] * Numeric['1] -> Numeric['1]
  добијен тип
  Numeric['6] * Numeric['6] -> Numeric['6]:

$$\text{Tip(izraz 9: n-1)} = \text{TipRes(-)} = \text{Numeric['6]} \qquad \text{(x.18)}$$
$$\text{Tip(izraz 10: n)} = \text{Numeric['6]} \qquad \text{(x.19)}$$
$$\text{Tip(izraz 11: 1)} = \text{Numeric['6]} \qquad \text{(x.20)}$$

- на основу правила простог израза на израз из хипотезе 19, где је име `n` аргумент функције:

$$\text{Tip(izraz 10: n)} = \text{'1} \qquad \text{(x.21)}$$

- на основу правила простог израза на израз из хипотезе 20, где је `1` константа типа Integer:

$$\text{Tip(izraz 11: 1)} = \text{Integer} \qquad \text{(x.22)}$$

- на основу правила простог израза и хипотезе 6, где је `1` константа типа Integer:

$$\text{Tip(izraz 4: 1)} = \text{Integer} \qquad \text{(x.23)}$$

Прављење скупа једначина

Ради прегледности, хипотезе су представљене груписане по левој страни:

Tip(`fact`)	= '1 -> '2	(x.1)
Tip(izraz 1: `if`...)	= '2	(x.2)
Tip(izraz 1: `if`...)	= TipRes(`if`) = '3	(x.3)
Tip(izraz 2: `n>1`)	= Bool	(x.4)
Tip (izraz 2: `n>1`)	= TipRes(>) = Bool	(x.7)
Tip(izraz 3: `n * fact(n-1)`)	= '3	(x.5)
Tip(izraz 3: `n * fact(n-1)`)	= TipRes(*) = Numeric['5]	(x.12)
Tip(izraz 4: `1`)	= '3	(x.6)
Tip(izraz 4: `1`)	= Integer	(x.23)
Tip(izraz 5: `n`)	= Value['4]	(x.8)
Tip(izraz 5: `n`)	= '1	(x.10)
Tip(izraz 6: `1`)	= Value['4]	(x.9)
Tip(izraz 6: `1`)	= Integer	(x.11)
Tip(izraz 7: `n`)	= Numeric['5]	(x.13)
Tip(izraz 7: `n`)	= '1	(x.15)
Tip(izraz 8: `fact(n-1)`)	= Numeric['5]	(x.14)
Tip(izraz 8: `fact(n-1)`)	= TipRes(`fact`) = '2	(x.16)
Tip(izraz 9: `n-1`)	= '1	(x.17)
Tip(izraz 9: `n-1`)	= TipRes(-) = Numeric['6]	(x.18)
Tip(izraz 10: `n`)	= Numeric['6]	(x.19)
Tip(izraz 10: `n`)	= '1	(x.21)
Tip(izraz 11: `1`)	= Numeric['6]	(x.20)
Tip(izraz 11: `1`)	= Integer	(x.22)

На основу парова хипотеза, које се односе на исте изразе, праве се договарајуће типовне једначине:

- на основу хипотеза 2 и 3:

  '2 = '3 (т.1)

- на основу хипотеза 4 и 7 добија се сагласност (нема једначине али је провера успешна):

  true

- на основу хипотеза 5 и 12:

  '3 = Numeric['5] (т.2)

- на основу хипотеза 6 и 23:

  '3 = Integer (т.3)

- на основу хипотеза 8 и 10:

  '1 = Value['4] (т.4)

- на основу хипотеза 9 и 11:

  Value['4] = Integer

  и одмах се закључује:

  '4 = Integer (т.5)

- на основу хипотеза 13 и 15:

  '1 = Numeric['5] (т.6)

- на основу хипотеза 14 и 16:

  '2 = Numeric['5] (т.7)

- на основу хипотеза 17 и 18:

  '1 = Numeric['6] (т.8)

- на основу хипотеза 19 и 21:

  '1 = Numeric['6]

  што је поновљена једначина и не евидентира се;

- на основу хипотеза 20 и 22:

  Numeric['6] = Integer

  и одмах се закључује:

  '6 = Integer (т.9)

Решавање система типовних једначина

Решава се добијени систем једначина тако што се примењују правила унификације типова. Најпре се прави таблица простих полиморфних типова:

'1 = ?
'2 = ?
'3 = ?
'4 = ?
'5 = ?
'6 = ?

Затим се на сваку од једначина и ту таблицу примењује унификација, при чему се у сваком кораку таблица по потреби ажурира:

- на основу т.1 и правила унификације два проста полиморфна типа, тип '3 се замењује типом '2:

  '3 = '2

- на основу т.2 и таблице се на основу правила унификације простих полиморфних типова, тип '2 се замењује типом Numeric['5]:

$$'2 = \text{Numeric}['5]$$

- на основу т.3 и таблице се добија:

$$'3 \rightarrow '2 \rightarrow \text{Numeric}['5] = \text{Integer}$$

па се на основу правила унификације комбинованих типова тип '5 замењује типом Integer:

$$'5 = \text{Integer}$$

- на основу т.4 и правила унификације простих полиморфних типова, тип '1 се замењује типом Value['4]:

$$'1 = \text{Value}['4]$$

- на основу т.5 и правила унификације простих полиморфних типова, тип '4 се замењује типом Integer:

$$'4 = \text{Integer}$$

- на основу т.6 и таблице се добија:

$$'1 \rightarrow \text{Value}['4] \rightarrow \text{Value}[\text{Integer}] = \text{Numeric}[\text{Integer}] \leftarrow \text{Numeric}['5]$$

па се на основу правила унификације два комбинована типа добија сагласност и ништа се не замењује;

- на основу т.7 и таблице се добија:

$$'2 \rightarrow \text{Numeric}['5] \rightarrow \text{Numeric}[\text{Integer}] = \text{Numeric}[\text{Integer}] \leftarrow \text{Numeric}['5]$$

па се на основу правила унификације два комбинована типа добија сагласност и ништа се не замењује;

- на основу т.8 и таблице се добија:

$$'1 \rightarrow \text{Value}['4] \rightarrow \text{Value}[\text{Integer}] = \text{Numeric}['6]$$

па се на основу правила унификације комбинованих типова тип '6 замењује типом Integer:

$$'6 = \text{Integer}$$

- на основу т.9 и таблице се добија:

$$'6 \rightarrow \text{Integer} = \text{Integer}$$

па се на основу правила унификације простих типова добија сагласност и ништа се не замењује.

На крају решавања система типовних једначина таблица има садржај:

$$'1 = \text{Value}['6]$$
$$'2 = \text{Numeric}['5]$$
$$'3 = '2$$
$$'4 = \text{Integer}$$

’5 = Integer
’6 = Integer

Упрошћавањем таблице се добија:

’1 = Integer
’2 = Integer
’3 = Integer
’4 = Integer
’5 = Integer
’6 = Integer

Нигде није дошло до противречности, што значи да су типови међусобно сагласни и да је провера типова успела. Заменом вредности типовних променљивих се добија да је тип функције `fact`:

’1 -> ’2 = Integer -> Integer

Индекс

Латинични називи и изрази

Литература

[Abra2001] David Abrahams, Aleksey Gurtovoy: **C++ Template Metaprogramming**, *Addison-Wesley Professional*, 2004.

[Alex2001] Andrei Alexandrescu: **Modern C++ Design**, *Addison-Wesley Professional*, 2001.

[ANSI:1978] **American National Standard Programming Language FORTRAN**, *ANSI(R) X3.9*, 1978.

[Atki1987] Malcolm P. Atkinson, O. Peter Buneman: **Types and Peristence in Database Programming Languages**, *ACM Computing Surveys, 19(2), 105-190*, 1987.

[Atki1989] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik: **The Object-Oriented Database System Manifesto**, у *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, 223-240, 1989.
(такође и у F. Bancilhon, C. Delobel, P. Kanellakis (eds.): ***Building an Object-Oriented Database System: The Story of O2***, Morgan Kaufmann, 1992.)

[Bach1964] Charles W. Bachman, S.B. Williams: **The Integrated Data Store – A general purpose programming system for random access memories**, *Proceedings of the AFIPS '64 (part I)*, 1964.

[Back1978] John Backus: **Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs**, *Communications of ACM*, 21(8), 613-641, 1978.

[Bare2000] Henk Barendregt, Eric Barendsen: **Introduction to Lambda Calculus**, 2000.
(*http://www.cs.ru.nl/E.Barendsen/onderwijs/sl2/materiaal/lambda.pdf*)

[Booth1996] Simon P. Booth, Simon B. Jones: **Are Ours Really Smaller Than Theirs?**, у Phil Trinder, ed., **Glasgow Workshop on Functional Programming**, *Ullapool, Scotland, July 1996. Department of Computing Science, University of Glasgow*, 1996.

[Bune1995] Peter Buneman, Shamim Naqvi, Val Tannen, Limsoon Wong: **Principles of Programming with Complex Objects and Collection Types**, *Theoretical Computer Science*, 149, 3-48, 1995.

[Card1985] Luca Cardeli, Peter Wegner: **On Understanding Types, Data Abstraction, and Polymorphism**, *Computing Surveys*, 17(4), 1985.

[Card1987] Luca Cardelli: **Basic Polymorphic Typechecking**, *Science of Computer Programming*, 8(2), 1987.

[Card1991] Luca Cardeli: **Typeful Programming**, у E. J. Neuhold, M. Paul, eds., **Formal Description of Programming Concepts**, *Springer-Verlag*, 1991.

[Card2004] Luca Cardeli: **Type Systems**, у Allen B. Tucker, *Computer Science Handbook, 2nd ed., Champan & Hall*, 2004.

[Codd1969] Edgar F. Codd: **Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks**, *IBM Research Report*, 1969.

[Codd1970] Edgar F. Codd: **A Relational Model of Data for Large Shared Data Banks**, *Communications of the ACM*, 377-387, 1970.

[Coll1994] Graham Collins: **Supporting Formal Reasoning About Standard ML**, Tech Report, *LFCS, University of Edinburgh*, 1994.

[Conn2004] Thomas Connolly, Carolyn Begg: **DataBase Systems: A Practical Approach to Design, Implementation and Management (4th Edition)**, *Addison-Wesley*, 2004.

[Cons1998] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.N. Volanschi, J. Lawall, J. Noye: **Partial Evaluation for Software Engineering**, *ACM Computing Surveys*, 30, 1998.

[Cous1987] G. Cousineau, P.-L. Curien, M. Mauny: **The Categorical Abstract Machine**, *Science of Computer Programming*, 8(2), 1987.

[Damas1982] Luis Damas, Robin Milner: **Principal Type-Schemes for Functional Programs**, *ACM SIGPLAN, Proc. 9th ACM SIGPLAN-SIGACT Sym. on Principles of Prog.Lan.*, 207-212, 1982.

[Darw1995] Hugh Darwen, C.J. Date: **The Third Manifesto**, *ACM SIGMOD Records*, 24, 39-49, 1995.

[Darw2001] Hugh Darwen: **Towards an Agreeable Model of Type Inheritance**, *Proc. British National Conference on Databases*: *Advances in Databases (BNCOD 2001), Rutherford Appleton Laboratory, UK, 09-11 Jul 2001, RAL Conference Proceedings, RAL-CONF-2001-003*, 2001.

[Date2003] C. J. Date: **An Introduction to Database Systems** (8th edition), *Addison-Wesley*, 2003.

[Date2007] C.J. Date, Hugh Darwen: **Databases, Types and the Relational Model, 3rd ed.**, *Addison-Wesley*, 2007.

[DCG:2009] Database Consulting Group LLC: **Dataphor.org**, 2009. (*http://dataphor.org/*)

[Dela2009] Kalen Delaney, Paul S. Randal, Kimberly L. Tripp, Conor Cunningham, Adam Machanic: **Microsoft SQL Server 2008 Internals**, *Microsoft Press*, 2009.

[Doug] Korry Douglas: PostgreSQL, 2nd ed,

[Fili2007] Andrzej Filinski: **On the relations between monadic semantics Source,** *Theoretical Computer Science*, 375(1-3), 41-75, 2007.

[Fowl2002] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford: **Patterns of Enterprise Application Architecture**, *Addison-Wesley*, 2002.

[Ghez1997] Carlo Ghezzi, Mehdi Jazayeri: **Programming Language Concepts (3rd edition)**, *John Wiley & Sons*, 1997. (ISBN 0-201-63455-4)

[Gilm1997] S. Gilmore: **Programming in Standard ML'97: A tutorial introduction**, Technical Report ECS-LFCS-97-364, *Laboratory for Foundations of Computer Science*, 1997.

[Giesl1995] Jurgen Giesl: **Termination Analysis for Functional Programs using Term Orderings**, *Pr. 2nd Int. Stat. Analysis Symp., LNCS 983*, Glasgow, Scotland, 1995.

[Gord2001] Andrew D. Gordon, Don Syme: **Typing a multi-language intermediate code**, *ACM SIGPLAN Notices*, 248-260, 2001.

[Hami1996] Graham Hamilton, Rick Cattell: **JDBC: A Java SQL API v.1.10**, *JavaSoft, Sun Microsystems, Inc.*, 1996.

[Harp2005] Robert Harper: **Programming in Standard ML**, *Carnegie Mellon University*, 2005.

[Heer2002] Bastiaan Heeren, Jurriaan Hage, Doaitse Swierstra: **Generalizing Hindley-Milner Type Inference Algorithms**, *Utrecht University, Tech.Rep. UU-CS-2002-31*, 2002.

[Hein1997] Nick Heinle: **JavaScript – When HTML Is Not Enough**, *WWW Journal, II/2 – "Scripting Languages"*, 1997.

[Hend1980] P. Henderson: **Functional Programming: Application and Implementation**, *Prentice Hall International*, 1980.

[Hudak1989] Paul Hudak: **Conception, Evolution, and Application of Functional Programming Languages**, *ACM Computing Surveys*, 21(3), 1989.

[Hudak1992] Paul Hudak, Simon Peyton Jones, Philip Wadler, eds.: **Report on the Programming Language Haskell**, *ACM SIGPLAN Notices*, 27(5), 1992.

[IBM:2008] **IBM DB2 Version 9.5: SQL Reference**, *IBM Corporation*, 2008.

[Ingr:2009] **Ingres Corporation Home Page**, Ingres Corporation, 2009. (*www.ingres.com*)

[ISO:1986] **ISO 9075:1987 Information technology – Database languages – SQL**, *ISO*, 1986.

[ISO:1999] **ISO/IEC 9075:1999 Information technology – Database languages – SQL**, *ISO/IEC*, 1999.

[ISO:2008] **ISO/IEC 9075:2008 Information technology – Database languages – SQL**, *ISO/IEC*, 2008.

[John1984] Thomas Johnsson: **Efficient Compilation of Lazy Evaluation**, y Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, *SIGPLAN Notices*, 19, 1984.

[Jones1987] Simon L. Peyton Jones: **The Implementation of Functional Programming Languages**, *Prentice Hall*, 1987.

[Jones1992] Simon L. Peyton Jones: **Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine**, *Journal of Functional Programming*, 2(2), 1992.

[Jones1993] Simon L. Peyton Jones, Philip Wadler: **Imperative Functional Programming**, *ACM Simposium on Principles of Programming Languages (POPL), Charlestown,* 1993.

[Kels1998] Richard Kelsey, William Clinger, Jonathan Rees: **Revised 5 report on the algorithmic language Scheme**, *ACM SIGPLAN Notices,* 33, 26-76, 1998.

[Lamb1987] David Alex Lamb: **IDL: sharing intermediate representations**, *ACM Transactions on Programming Languages and Systems,* 9(3), 297-318, 1987.

[Lans2007] Rick F. van der Lans: **SQL for MySQL Developers: A Comprehensive Tutorial and Reference**, *Addison-Wesley Professional*, 2007.

[Lee1998] Oukesh Lee, Kwangkeun Yi:**Proofs About a Folklore Let-Polymorphic Type Inference Algorithm**, *ACM Transactions on Programming Languages and Systems,* 20(4), 707-723, 1998.

[Leij1999] Daan Leijen, Erik Meijer, **Domain Specific Embeded Compilers**, *2nd USENIX Conference on Domain-Specific Languages (DSL), Austin, USA, October*, 1999.

[Malk2002] Saša Malkov: ***Wafl* – funkcionalni programski jezik za razvoj Veb aplikacija**, magistarski rad, *Matematički fakultet Univerziteta u Beogradu,* 2002.

[Malk2007] Saša Malkov, Andrija Antonijević, et.al.: **Informacioni sistem fakulteta StudInfo**, tehnička dokumentacija, *Matematički fakultet Univerziteta u Beogradu,* 2007.

[Malk2008a] Saša Malkov, Nenad Mitić, Žarko Mijajlović: **Nikola Tesla Online Clipping Library Prototype**, *Review of the National Center for Digitization*, 12, 75-81, 2008. (*http://virlib.matf.bg.ac.yu/tesla/index.wafl*)

[Malk2008b] Saša Malkov: **On Photograph Library Design – Groman Library**, *Review of the National Center for Digitization*, 13, 27-36, 2008.

[Malk2008c] Saša Malkov, Miodrag Živković, Miloš Beljanski, Michael Hall, Snežana Zarić: **A Reexamination of Correlations of Amino Acids with Particular Secondary Structure – Classification of Amino Acids Based on Their Chemical Structure**, *Journal of Molecular Modeling*, 14(8), 769-775, 2008.

[Malk2009] Saša Malkov, Miodrag Živković, Miloš Beljanski, Srđan Stojanović, Snežana Zarić: **A Reexamination of Correlations of Amino Acids with Particular Secondary Structures**, The Protein Journal, 74-86, 2009.

[Mats2008] David Flanagan, Yukihiro Matsumoto: **The Ruby Programming Language**, *O'Reilly Media, Inc.*, 2008.

[McCa1960] J. McCarthy: **Recursive Functions of Symbolic Expressions and Their Computation By Machine, Part I**, *Communications of ACM,* 3(4), 184-195, 1960.

[McCa1978] J. McCarthy: **History of Lisp**, y **Preprints of Proceedings of ACM SIGPLAN History of Programming Languages Conference**, *SIGPLAN Notices,* 13, 1978.

[Melt2002] Jim Melton: **Advanced SQL: 1999 - Understanding Object-Relational and Other Advanced Features**, *Morgan Kaufmann*, 2002.

[Meltz2005] Dean Meltz, Rick Long, Mark Harrington, Robert Hain, Geoff Nicholls: **An Introduction to IMS: Your Complete Guide to IBM's Information Management System**, *IBM Press*, 2005.

[Miln1978] R. Milner: **A Theory of Type Polymorphism in Programming**, *Journal of Computer and System Science*, 17, 1978.

[Miln1997] Robin Milner, Mads Tofte, Robert Harper, David MacQueen: **The Definition of Standard ML (Revised)**, *MIT Press*, 1997.

[Mitić1995] Nenad Mitić: **Funkcijski interfejs ka relacionim bazama podataka i primene**, doktorska disertacija, *Matematički fakultet Univerziteta u Beogradu*, 1995.

[Miti2003] Nenad Mitić, Saša Malkov, Neda Bokan, Zoran Rakic, et al: **Online Geometry Book „Analytic Geometry"**, *University of Belgrade – Faculty of Mathematics*, 2003. (*http://codd.matf.bg.ac.yu/ang/pocetak.wafl*)

[Mooc:2008] **Moochikit Home Page**, 2008. (*http://mochikit.com*)

[Moun1999] Jon Mountjoy: **The Spineless Tagless G-machine, naturally**, *ACM Sigplan Notices*, 34(1), 1999.

[MS.MIDL] **MIDL Language Reference**, *Microsoft* (*http://msdn.microsoft.com/en-us/library/aa367091.aspx*)

[MS.COM] **COM: Component Object Model Technologies**, *Microsoft* (*http://www.microsoft.com/com*)

[MS.NET] **.NET Framework Overview**, *Microsoft* (*http://www.microsoft.com/net*)

[Moon1979] David Moon: **Maclisp Reference Manual**, 1979. (*http://zane.brouhaha.com/~healyzh/doc/lisp.doc.txt*)

[Geig1995] Kyle Geiger: **Inside ODBC**, *Microsoft Press*, 1995.

[Oksa2001] Kenneth Oksanen: **A Replicated and Persistent Functional Programming Environment**, *Dept. of Comp.Sci., Fac. of Inf.Tech., Helsinki University of Technology*, 2001.

[OMG:2002] **The Common Object Request Broker: Architecture and Specification, version 3**, *Object Management Group*, 2002.

[Ora:2002] **Oracle 9i: SQLJ Developer's Guide and Reference, Release 2**, *Oracle, A96655-01*, 2002.

[Pavl1996] Gordana Pavlović-Lažetić: **Osnove relacionih baza podataka**, *VESTA i Matematički fakultet, Beograd*, 1996.

[Post:2007] The PostgreSQL Global Development Group: **PostgreSQL Reference Manual - Volume 1 SQL Language Reference**, *Network Theory Ltd*, 2007.

[Puce2001] Riccardo Pucella: **Notes on Programming Stadard ML of New Jersey**, *Dept. of Computer Science, Cornell University*, 2001. (*http://www.smlnj.org*)

[Rama2002] R.Ramakrishnan, J.Gehrke: **Database Management Systems, 3rd ed.**, *McGraw-Hill Professional*, 2002.

[Reed1978] D.P. Reed: **Naming and Synchronization in a Decentralized Computer System**, *MIT dissertation*, 1978.

[REL:2009] Dave Voorhis: **An Implementation of Tutorial D database language**, 2009. (*http://dbappbuilder.sourceforge.net/Rel.php*)

[Robi1965] J.A.Robinson: **A machine-oriented logic based on the resolution principle**, *Journal of ACM*, 12, 23-41, 1965.

[Ross2003] Guido van Rossum: **An Introduction to Python**, *Network Theory Ltd.*, 2003.

[SAP:2009] **The Complete SAP MaxDB Documentation**, *SAP*, 2009. (*http://maxdb.sap.com/documentation*)

[Stall1984] Richard Stallman, Daniel Weinreb, David Moon: **Lisp Machine manual, 6th ed.**, 1984. (*http://common-lisp.net/project/bknr/static/lmman/frontpage.html*)

[Silva2006] Alexandra Silva, Joost Visser: **Strong Types for Relational Databases**, *Proceedings of the ACM SIGPLAN workshop Haskell'06, Portland, Oregon, USA*, 2006.

[Stee1993] Guy L. Steele, Richard P. Gabriel: **The evolution of Lisp**, *ACM SIGPLAN Notices*, 231-270, 1993.

[Teit1974] Warren Teitelman: **Interlisp Reference Manual**, Xerox, Palo Alto Research Center, 1974. (*http://bitsavers.org/pdf/xerox/interlisp/1974_InterlispRefMan.pdf*)

[Thom1999] Simon Thompson: **Type Theory & Functional Programming**, *Computing Laboratory, university of Kent*, 1999.

[Tolm2003] Andrew P. Tolmach, Sergio Antoy: **A monadic semantics for core Curry**, *Electronic Notes in Theoretical Computer Science*, 86(3), 2003.

[Turn1979] D. A. Turner: **A new implementation technique for applicative languages**, *Software Practice and Experience*, 9, 1979.

[W3C:2007] **Web Services Description Language (WSDL) Version 2.0**, *W3C Recommendation*, 2007. (*http://www.w3.org/TR/wsdl20/*)

[Wong2000] Limsoon Wong: **Kleisli, a Functional Query System**, *J. of Functional Programming*, 10(1), 19-56, 2000.

[Yaof2005] Yaofei Chen, R. Dios, A. Mili, Lan Wu, Kefei Wang: **An empirical study of programming language trends**, IEEE Software, 22(3), 72-79, 2005.

[Živk2006] Miodrag Živkovic, Saša Malkov, Snežana Zarić, Milena Vujošević-Janičić, Jelena Tomašević, Goran Predović, Novica Blažić, Miloš Beljanski: **Statistical Dependence of Protein Secondary Structure on Amino Acid Bigrams**, *Chemical Industry & Chemical Engineering Quarterly*, 12, 82-85, 2006.