МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ІНСТИТУТ ПІСЛЯДИПЛОМНОГО НАВЧАННЯ
Кафедра інженерії програмного забезпечення

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
Сидоров М.О.
„____”_____2013р.

ДИПЛОМНИЙ ПРОЕКТ

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬО-КВАЛІФІКАЦІЙНОГО РІВНЯ
„СПЕЦІАЛІСТ ”

Тема: Засіб керування оптимізацією програм

Виконав: Мельниченко В. А.

Керівник: к.т.н., доц. Авраменко О.А

Нормоконтролер к.т.н., доц. Радішевський М.Ф.

Київ-2013

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Інститут післядипломної освіти

Кафедра інженерії програмного забезпечення

Напрям (спеціальність): 7.05010301 «Програмне забезпечення систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Сидоров М.О.

“____”_____2013 р.

ЗАВДАННЯ

на виконання дипломного проекту

Мельниченка Володимира Анатолійовича
(призвіще, ім’я, по батькові)

1. Тема дипломного проекту: «Засіб керування оптимізацією програм»
затверджена наказом ректора від “04” березня 2013 р. № 487/ст
2. Термін виконання проекту: з 25.03.13 по 23.06.13
3. Вихідні данні до проекту: програмний засіб виконання оптимізацій використовуючи анотації з вихідного коду програми
4. Зміст пояснювальної записки (перелік питань, що підлягають розробці): аналіз існуючих технік оптимізації; техніка керування оптимізацією програм; реалізація засобу керування оптимізацією програм
5. Перелік обов’язкового графічного (ілюстрованого) матеріалу:
Структурна схема програмного засобу; Схема спільної роботи компонентів; Алгоритм обробки анотацій; Види сумісного користування даними; Приклад використання відкладених обчислень; Анотації

Календарний план-графік

№пор.	Етапи виконання дипломного проекту	Термін виконання	Примітка
1	Ознайомлення з літературою	25.03.2013 - 10.04.2013	
2	Робота над розділом «Аналіз існуючих технік оптимізації»	10.04.2013 - 23.04.2013	
3	Робота над розділом «Техніка керування оптимізацією програм»	23.04.2013 - 07.05.2013	
4	Розробка логічної програми	07.05.2013 - 15.05.2013	
5	Розробка засобу керування оптимізацією програм	15.05.2013 - 02.06.2013	
6	Оформлення пояснювальної записки дипломної роботи	02.06.2013 - 12.06.2013	
7	Підготовка графічних матеріалів	12.06.2013 - 20.06.2013	
8	Підготовка доповіді для захисту дипломної роботи	20.06.2013 - 25.06.2013	

Дата видачі завдання « 25 » березня 2013р.

Керівник дипломної роботи Авраменко О.А.

Завдання прийняв до виконання Мельниченко В. А.

РЕФЕРАТ

Пояснювальна записка до дипломного проекту " Засіб керування оптимізацією програм": 83 с., 11 рис., 3 табл., 2 додатка, 17 літературних джерел.

АНОТАЦІЇ, АВТОМАТИЧНІ РОЗМІРКОВУВАННЯ, МІНМАЛЬНА МОДЕЛЬ, ПРЕДМЕТНО-ОРІЄНТОВАНІ ОПТИМІЗАЦІЇ, МЕТАОБЧИСЛЕННЯ, ПРИЙНЯТТЯ РІШЕНЬ, ФУНКЦІОНАЛЬНІ ПРОГРАМИ, LLVM, CLASP, ASP,.

Об'єкт розробки — оптимізація програм на функціональних мовах програмування

Мета роботи – підвищення ефективності роботи програмного забезпечення шляхом впровадження додаткових оптимізацій

Метод дослідження — спостереження за результатами впливу різних типів оптимізації

Встановлено, що цілу низку технік оптимізацій, що зазвичай виконуються вручну, можно замінити авто-перетвореннями вихідного кода, також автоматично визначаючи контексти, в яких їх застосування є корисним,

Результати дипломного проектування: рекомендується використовувати при реалізації складних систем, що складаються з багатьох незалежних компонентів.

Прогнозні припущення щодо розвитку об'єкта дослідження — використання дослідженої методики у поєднанні з даними динамічного аналізу, визначая характеристики поведінки програми у різних станах. Крім того, перспективно використовувати методику при оцінюванні метрик програми та якості архітектури.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ЛП — логічна програма

ПЗ — програмне забеспечення

ПП — породжуюче программування

ФВП – функції вищого порядку

ФМ — функціональні мови програмування

ASP — Answer Set Programming

ВСТУП

У даній роботі розглядаються методи ефективного застосування високорівневих оптимізацій програмного коду, і крім того, важливою метою є якісне роспізнавання ситуацій, коли ці оптимізації потрібно застосовувати.

Суть високорівневих оптимізацій полягає в тому, щоб автоматично підтримувати два різних представлення однієї і теж програми. Це власне сам вихідний код програми, який составляє програмист, намагаючись зробити його наочним, вираженим у термінах проблемної області, узагальненим, структурованим тощо, а інше представлення — це код, отриманий у результаті автоматичних перетворень, який зберігає лише еквівалентність по відношенню до оригінального коду, проте більш ефективний і придатний до компіляції. Це представлення в багатьох аспектах більш конкретизоване, воно можливо містить інформацію о явним керуванні пам'яттю та іншими ресурсами, тим самим звільнючи програміста від зайвої роботи.

Основна ідея полягає в тому, щоб конструювати це представлення не на загальних підставах, а навпаки, використовуючи усю доступную інформацію, будувати его спеціалізуючи для конкретного контексту виконання. Усю потрібну для цього інформацію автоматично не вдається зібрати, тому в даній роботі розглядаються способи допомогти автоматичному аналізу, надавая додаткову інформацію у вигляді анотацій, що вносить програміст до вихідного коду.

У розділі I розглядаються теоретичні обґрунтування, у розділі II розглядаються принципи конкретних видів перетворень вихідного коду, а у розділі III наведений опис реалізації виконання перетворень на підставі обробки анотацій.

РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ ТЕХНІК ОПТИМІЗАЦІЇ ПРОГРАМ

1. 1. Типи оптимізацій

Оптимізація - це спосіб перетворення програми з метою одержання більш оптимального програмного коду, зберігаючи його функціональні можливості.

Таке перетворення покращує «якість» коду. Як правило, під «покращенням» розуміють більш швидке виконання (або запуск) або зменшення вимог до пам'яті. Крім того, у зв'язку з широким розповсюдженням портативних пристроїв, метою може ставитися використання програмою меншої кількості енергії.

У сучасних компіляторах виділяється окрема фаза для оптимізації програми. У цій фазі автоматично проводиться маса спеціалізованих технік аналізу і перетворення, сконцентрованих на різних аспектах програми і проводяться вони на різних рівнях.

Зазвичай виділяють групи автоматичних, машинних технік оптимізації такі як: локальні, що проводяться в рамках одного виразу або всередині однієї процедури; процедурні, що проводяться на рівні процедури, наприклад оптимізація хвостовій рекурсії тощо.; міжпроцедурні, що аналізують відразу декілька процедур у зв'язці.

Тим не менше, всі ці операції є низькорівневими в тому сенсі, що вони працюють з цілком певним, заданим набором абстракцій: блоки, шляхи виконання, цикли, умови, виклики функцій.

У зв'язку з цим, незважаючи на різноманітність застосовуваних технік, на добре розроблені та ефективні способи аналізу та їх реалізації, програмісту тим не менш доводиться вручну додатково оптимізувати програму, вносячи поліпшення, пов'язані з специфічними для цієї програми абстракціями, відповідними до предметної області.

Обмеження, властиві ручній оптимізації програм

У зв'язку з досить швидким здешевленням обчислювальної техніки, укупі із зростанням обчислювальних потужностей і, додатково, з розвитком перспектив «хмарних» обчислень, в багатьох мовах програмування і парадигмах розробки ПЗ проявляється певна тенденція. Від-ходить зміщення акценту з розробки швидких і ефективних програм у бік підвищення наочності коду, забезпечення безпеки, легкості підтримки, модульності, і можливості незалежної розробки окремих компонентів і можливо найголовніше - швидкості розробки програм.У деяких випадках прагнення виробляти ефективний вихідний код вступає в протиріччя з іншими цілями, що стоять перед командою розробників.

Як видно з рис. 1.1, часто ефективністю вихідного коду жертвують, наприклад, на догоду швидкого циклу розробки. Диаграма відображає реальну ситуацію при розробці проекта, коли існує багато часто суперечливих цілей та обмежень, що треба дотримати. Багато методологій розробки ПО багато уваги приділяють далеким від оптимізації цілям.

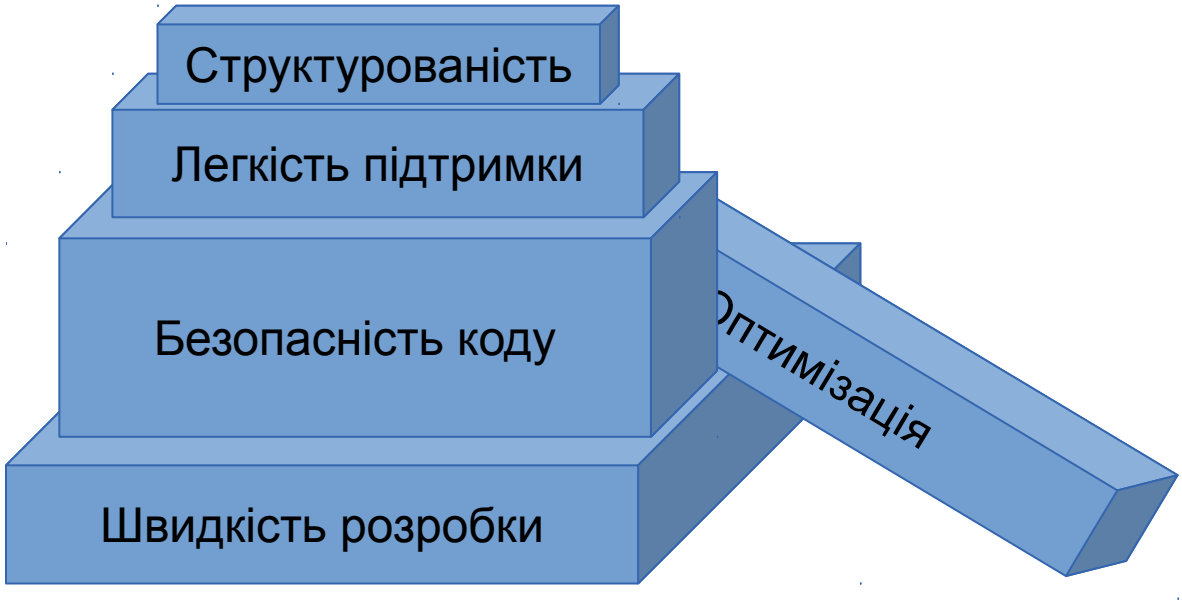

Рис 1.1. Наочне уявлення про типові протиріччя між цілями, що стоять перед командою розробників програмного проекту.

Виходячи з цих причин, пріоритетність оптимізації кінцевого програмного продукту знижується, і нижче будуть розглянуті окремі ситуації, де це особливо проявляється.

- *Високорівневі мови.* Високорівневі мови завойовують все більшу популярність. Їх основна перевага - приховування від розробника окремих аспектів реалізації програми і компонентів, і відповідно, вони дають можливість зосередитися на вирішенні власне поставленого завдання. Натомість, ці мови жертвують продуктивністю, т. к. змушені проводити перевірки коректності програми вже в стадії виконання (у разі динамічних мов), або вибирають часто вже не ефективні, задані «за замовчуванням» стратегії роботи тих чи інших компонентів (функціональном мови). Високорівневі мови часто вибирають для прототипування, складання «першої версії» програми, тому що розробка ви-

робляється набагато швидше і простіше. Тому високорівневими мовами користується безліч непрограмістів: інженерів, вчених, дослідників та інш. - Охочих не витрачаючи багато зусиль скласти програму для своїх потреб, проводячи розрахунки або автоматизуючи свої дії на комп'ютері. Відповідно, питань оптимізації приділяється мало уваги, тому що такі мови просто не надають можливості ретельно оптимізувати код, і з іншого боку не-професійні користувачі просто не володіють достатнім обсягом спеціальних знань для цього.

- *Рефакторинг та підтримка.* При оптимізації програми найчастіше порушуються багато архітектурних принципів, погіршується ясність, наочність і «прозорість» коду. Так як багато технік оптимізації засновані на інформації про внутрішні деталі реалізації об'єктів та їх взаємодії один з одним, то на етапі оптимізації їх поведінку і реалізація жорстко фіксуються, з тим, щоб домогтися спільного використання структур даних і внутрішніх алгоритмів, а також для інтенсивного кешування. Виходячи з цього, оптимізований код важко підтримувати, практично не можливо вносити зміни і проводити рефакторінг. Як наслідок, фазу оптимізацію залишають наостанок, як останню фазу розробки, виконувану, коли програма вже написана і працює.Тому, для програм які протягом усього свого життєвого циклу постійно піддаються змінам, така фаза взагалі часто не настає.

У цих, і багатьох інших ситуаціях застосування «класичної» оптимізації, що проводиться вручну, ускладнено, і на допомогу можуть прийти більш «інтелектуальні» методи автоматичної оптимізації, що враховують використовувані в конкретній програмі нестандартні абстракції і специфіку предметної області.

1.2. Оптимізація функціональних мов

1.2.1. Переваги використання функціональних мов

Одне із значних місць у дослідженнях з теоретичного програмування займає функціональне програмування. Ведуться протягом уже трьох десятиліть розробки в цій області останнім часом мають стійку тенденцію до розширення як в академічних, так і в промислових організаціях.

Виконання програми на функціональній мові, кажучи неформально, полягає у виклику функції, аргументами якої є результати інших функцій (функції першого порядку) чи власне функції (функції вищого порядку), які в свою чергу також можуть бути суперпозиціями в загальному випадку довільної глибини.

З точки зору програміста, основна частина програми складається з сукупності визначень функцій, як правило, рекурсивних. Така особливість притягальна тим, що отримувані програми є свого роду ієрархічними специфікаціями.

Приклад знаходження списку з порядком елементів зворотним до заданому:

```
reverse' [] = []
reverse' [x] = [x]
reverse' xs = last xs : reverse' (init xs)
```

Тут вказується скоріше не точний алгоритм побудови зворотнього списку, а відносини між прямим і зворотнім списками. Крім того, в явному вигляді вказані часткові випадки, що дає більше інформації для, наприклад, інструментів тестування.

Внаслідок того, що функціональна парадигма програмування є різновидом і природною реалізацією декларативного підходу, такого важливого на рі-

вні архітектури ПЗ, можна очікувати, що об'єднання двох споріднених технологій дасть додаткові синергетичні переваги .

Нижче будуть показані деякі особливості використання в цьому зрізі саме функціонального підходу.

Структурність

Так як ПЗ стає все більш і більш складним, потрібно приділяти все більшу увагу забезпеченню якісного структурування вихідного коду.

Очевидно, що добре структуровані програми легко писати, налагоджувати, удосконалювати і підтримувати, крім того, це дає можливість оформити колекцію незалежних компонентів, які можуть бути згодом повторно використані в інших проектах, зменшуючи майбутні витрати.

Багато дослідників погоджуються з тим, що ФМП надають в чомусь більш послідовну і кращу підтримку модульності, ніж імперативні мови.

Приклад алгоритму скалярного добутку двох векторів:

```
dotProduct :: [a]->[a]->a
dotProduct  = (sum .) . zipWith (*)
```

Таким чином, нова операція повністю визначена у вигляді супер-позиції вже існуючих - це прояв здатності гнучко комбінувати вже існуючі функції для досягнення потрібного результату.

У рамках імперативного підходу виникає необхідність розробляються абстракції і ділити програму на компоненти, які обмежені взаємодією один з одним в рамках спочатку заданих абстракцій. На відміну від цього, два інструменти з арсеналу ФМП зокрема пропонують особливий внесок у проблему забезпечення модульності: це функції вищого порядку (ФВП) і відкладені обчислення.

ФВП пропонують можливість легко абстрагувати необхідну частину коду створивши функцію вищого порядку, роблячи програму більш декларативною і зрозумілою.

Відкладені обчислення дають можливість оперувати «нескінченними» структурами даних, пропонуючи потужну техніку і своєрідні додаткові способи декомпозиції програм, дозволяючи, наприклад, ізолювати «генератори» даних від «споживачів» даних.

Деякі дослідники вважають відкладені обчислення найбільш потужним і перспективним способом забезпечення модульності і інкапсуляції.

Приклад використання відкладених обчислень у вигляді функції повертаючої чотири найменших ел-та у списку:

```
topFour:: [Int]->[Int]
topFour list = take 4  (sort list)
```

Цей приклад демонструє логічне відділення коду, який виробляє дані (генератора, в даному випадку - функції sort, від коду, який цими даними користується, в даному випадку функції (take 4), яка використовує лише перші 4 елементи отриманого списку. «Ледачий» обчислень виявляється в тому, що не виробляється повної сортування списку, а тільки в обсязі достатньому для знаходження потрібних 4 елементів.

Таким чином можна писати більш абстрактні програми, не побоюючись падіння продуктивності.

Перенесення і використання цієї техніки в нефункціональних мовах досить ускладнено, через те, що відкладені обчислення базуються на невизначеному порядку виконання, які, очевидно призведуть до непередбачуваних наслідків для функцій залежних (і впливаючих) на якийсь стан, контекст виконання.

Декларативність

Насамперед ця властивість випливає з притаманною визначенню функцій декларативності, що дозволяє писати програми в термінах того, що треба робити, а не як це робиться.

У цьому аспекті важливо згадати властивість «чистоти» (referential transparency). Це властивість позначає, що якщо в тексті програми, припустимо зустрічається визначення y = f (x), тоді скрізь, де зустрічається «y», цю змінну можна синтаксично замінити на f (x) без втрати працездатності алгоритму.

Невизначеність в порядку обчислень і «чистота» спонукають представляти алгоритм у декларативному вигляді, у вигляді зв'язку між даними і формального опису завдання замість занадто жорстко заданого конкретного алгоритму і порядку обчислень.

Саме тому функціональні мови розглядаються як засоби декларативного програмування.

Для прикладу, нижче наведено програма вирішення простої системи квадратних нерівностей:

```
roots:: [Int]
roots = [x | x <-[0..], x^2 > 3, x^2 < 11]
```

Ця програма повертає список рішень і демонструє декларативність написання: зазначена область допустимих значень (ОДЗ) коренів системи і виписані нерівності.

Використання «контрактів»

«Контракт» - це перелік умов, які компоненти висувають один іншому для збільшення надійності і сумісності при спільній роботі. Як вже було

сказано, програму на ФМП можна розглядають як перелік специфікацій, що часто однозначно визначає необхідні контракти.

Додатково, можливість використовувати «порівняння за зразком» дозволяє гарантувати обов'язкову перевірку зазначеного таким чином «контракту». Варто додати, що «порівняння за зразком» має аналогії з «рефлексією» - технікою, що має важливе значення в сучасному проектуванні

Наочність

Можливо найбільш формальним визначенням того, що таке наочність програми – здатність у вихідному коді висловити «наміри» програміста, що є основним акцентом в «інтенсіональном» програмуванні.

Оформлення програми у вигляді комбінації деяких функцій вищого порядку укупі з функціями, що визначають особливі спеціалізації для вирішення необхідної задачі, надає зручні способи висловити довільні абстрактні принципи та ідеї закладені в реалізації багатьох алгоритмів.

Тут простежується аналогія з «розширюваними» мовами, фактично ФМП дозволяють розширювати синтаксис новими конструкціями управління, там де це прийнятно.

Описаний підхід, а також декларативна природа програм і використання «контрактів» дають досить багато можливостей для написання наочного коду.

Легкість конфігурування (параметризація)

Передача в якості аргументу функції іншої функції надає можливість гнучкого налаштування роботи та поведінки деякого компоненту. Часто такі аргументи називають «стратегіями», керуючими окремими аспектами роботи компонента.

Короткий приклад, який ілюструє цей принцип проявляється наприклад в на-

ступній програмі, що повертає список тільки простих чисел:

```
onlyPrimes:: [Int]
onlyPrimes = filter  isPrime [1..100]
```

Тут filter це функція вищого порядку, фільтруюча список по заданому предикату. Фактично предикат, в даному випадку isPrime є стратегією, що вказує конкретну поведінку компонента.

Формальні перетворення

Функціональні програми придатні для формального аналізу та маніпулювання.

Внаслідок того, що ми можемо звернутися до звичайного математичному апарату, т. к. виходячи з властивості «чистоти» можна замінювати еквівалентні вирази, знаючи що алгоритм не згубить працездатність через некоректні «побічні» ефекти, формальне маніпулювання функціональними програмами виконується відносно просто і при встановленні їх властивостей (верифікація), і при перетворенні програм у більш ефективні форми, з метою їх оптимізації.

Надійність

Як вже зазначалося, властивість «чистоти» і редукована семантика функціональних мов дозволяють значно легше проводити формальні докази коректності алгоритмів. У той час як імперативні мови вимагають розмірковування в розширених численнях, такі як предикати Дейкстри або найслабші передумови Хоара.

Декларативність і наочність, використання «контрактів» і веріфіцірує-мость - є найважливішими передумовами для забезпечення надійності ПЗ.

Паралелізм

Дуже важлива властивість. Через відсутність "побічних ефектів" програми написані на ФМП, часто навіть без додаткових зусиль програміста природним чином можна распаралелівать. З урахуванням сучасних багатопроцесорних і кластерних архітектур апаратного забезпечення, існує гостра потреба в мовах з потенціалом до автоматичного планування паралельного виконання програм.

Оптимізація

Відкладені обчислення дозволяють уникати непотрібних обчислень, зациклення або помилок, пов'язаних з роботою зі складними залежностями між даними і обчислюючи тільки те, що потрібно в конкретній ситуації. Гарантована елімінація «зайвих» обчислень дозволяє спрощувати складність алгоритмів.

При такому підході обчислювальна складність залежить вже не стільки від природи алгоритму, скільки від його використання, від того, які дані і в якому порядку необхідно отримати.

Кожна з цих властивостей обумовлено притаманною функціональним мовам математичною природою: конструкції функціональних програм є просто функціями, математичними функціями, що описують перетворення вхідних значень у вихідні і не стосуються контексту, в якому вони застосовуються.

Все це робить функціональне програмування вельми привабливим як у теоретичному, так і в практичному аспектах. Тому, в останні роки ведеться велика кількість методів розробки та аналізу програмного забеспечення на функціональних мовах.

1.2.2. Оптимізуючі метаобчислення для програм на функціональних мовах

Метаобчислення - це розділ програмування, присвячений розробці методів аналізу та перетворення програм за рахунок реалізації конструктивних метасистем (метапрограм) над програмами. У метаобчисленні в першу чергу включають теорію суперкомпіляції та близькі методи і засоби. Приставка "мета" вказує на те, що програма в метаобчисленнях розглядається як об'єкт аналізу та / або перетворення. Таким чином, метапрограми працюють над іншими програмами, контролюють, аналізують або імітують їх роботу [5, 6].

До основних, фундаментальних метаобчисленням зазвичай відносять:

- Спеціалізація програм
- Композиція програм
- Інверсія програм

Спеціалізація програм — це породження за універсальною програмою з низкою параметрів спеціалізованої програми, коли значення частини параметрів відомі і фіксовані.

Формально, це можна записати у вигляді: $f(x,y) \rightarrow f_A(y) = f(A,y)$, де $f_A(y)$ спеціалізована програма приймаюча на вхід один аргумент .

При спеціалізації відома інформація поширюється по тексту програми і використовується для її оптимізації та підняття ефективності в багато разів. Практичну користь від хороших засобів спеціалізації неможливо переоцінити. Програмісти багаторазово створюють версії програм на різні випадки життя через те, що універсальні програми не влаштовують їх по ефективності. Більше того, виявляється, самовикористання спеціалізатора дозволяє вирішувати незвичайні завдання, наприклад, перетворення інтерпретатора в компіля-

тор.

Композиція програм — це породження за двома (або декількома) підпрограмам, передавальним результати однієї в якості аргументів іншої, більш ефективної програми.

Формально композицію можна виразити таким чином:

$f(x), f(y) \rightarrow f_g(x) = f(g(x))$, де $f_g(x)$ - композитна програма

Композитна програма або виконує ту ж роботу взагалі без формування проміжних даних («за один прохід»), або містить оптимізовані версії вихідних підпрограм з урахуванням того, що вони працюють в контексті один одного. Композиція програм є узагальненням спеціалізації, оскільки

$f_A(y) = f(g(y), y) = f(g(x)) = f_g(x)$, де $g(y) = A$.

Ефекти від її застосування були б ще більш різноманітні, ніж від спеціалізації.

Инверсія програм — алгоритм, за програмою, що реалізує деяку функцію, що породжує програму, що реалізовує зворотну функцію, тобто за значенням першої програми видає відповідний аргумент.

Формально виражається наступною формулою:

$f(x), y = f(x) \rightarrow f_{-1}(y) = x$.

Цінність інверсії програм у тому, що дуже часто зворотний функцію задати набагато легше, ніж ту, яку потрібно запрограмувати. Повноцінна інверсія програм - дуже складне завдання; зараз вона успішно вирішується лише для деяких класів програм. Інверсія фактично лежить в основі логічного програмування.

Крім цих трьох завдань можна запропонувати ще багато операцій над програмами, які хотілося б виконувати. Але дослідження показують, що багато з них несподівано зводяться до цих трьох, їх комбінаціям, взаємною за-

стосуванню і самовикористанню. Тому вони є головними тестовими завданнями метаобчислень. Прогрес за методами їх рішення відразу дасть результати і для інших завдань.

Одна з найбільш перспективних ідей в теорії метаобчислень є суперкомпіляція.

Суперкомпіляція - як підвид метаобчислень, заснована на метасистемному переході, тобто переході від самої програми до її узагальненої версії, де замість конкретних параметрів вказані змінні, кіт. можуть приймати будь-яке значення. У термінології теорії суперкомпіляції такі змінні, а також описувані ними безлічі можливих значень називаються конфігураціями.

Суперкомпілятор намагається виконати таку узагальнену програму, незважаючи на те, що в ній не містяться конкретні параметри. При досягненні, наприклад, керуючих конструкцій, коли необхідно вибрати ту чи іншу гілку подальшого виконання залежно від конкретного значення умови, метаобчислення проводяться паралельно у всіх можливих розгалуженнях.

Процес обчислення програми на будь-якій мові - це покрокові перетворення станів абстрактної машини-виконавиці цієї мови. Правила цих перетворень визначають семантику мови. Послідовність станів, які проходить процес, утворюють шлях обчислень.

За наявності в узагальненій програмі керуючих конструкцій, породжується потенційно нескінченне дерево всіх можливих шляхів ви-чисельний. Де коренем такого дерева є початкова конфігурація, а кожна вершина - можлива конфігурація на певному етапі ви-чисельний. Таке дерево ще називають деревом процесів, підкреслюючи той факт, що процесам обчислень відповідають шляху в дереві від кореня до будь-якої іншої вершини.

Так-як дерево процесів будується так, щоб конфігурації включали в себе всі можливі стани, а дуги враховували всі можливі розгалуження, то завдяки цьому можна сказати, що дерево процесів «містить у собі» всі можливі

процеси обчислень.

Таким чином, виявляється, що дерево процесів містить у собі всю необхідну інформацію для обчислень, і, щоб виконати процес, вже не потрібно звертатися до вихідної програмі. Це означає, що дерево процесів можна інтерпретувати як програму, еквівалентну вихідної.

У дерева процесів лише один недолік - у загальному випадку воно нескінченно. Тому проводяться спеціальні кроки щодо його перетворення, такі як: зациклення, розсічення конфігурацій і узагальнення конфігурацій. В результаті дерево перетвориться в кінцевий граф. Причому, кінцівку графа досягається за рахунок зациклення, а розсічення та узагальнення допомагають приводити конфігурації до виду, придатному для зациклення.

Одержаний кінцевий граф конфігурацій і є залишковою програмою, еквівалентної вихідної і є результатом суперкомпіляції.

Можливість згорнути потенційно нескінченне дерево в кінцевий граф, залежить від якості узагальнень конфігурацій.

Узагальнення - це найскладніша частина методу суперкомпіляції і основний напрямок досліджень. Запропоновано декілька цікавих і практичних методів узагальнення, але тема далеко не закрита і не буде ніколи закрита, оскільки має справу з алгоритмічно нерозв'язним завданням.

Таким чином, метавичісленія, і зокрема суперкомпіляція - дозволяють проводити нетривіальні оптимізують перетворення програм. Важливою відмінністю від інших видів оптимізації є можливість виокремлювати з вихідного коду програми додаткову інформацію, спиратися при оптимізації на *семантику* програми.

У теорії метаобчислень досягнуті лише часткові успіхи, т. к. ряд завдань в загальному випадку алгоритмічно нерозв'язний, тому повністю автоматичне проведення метаобчислень можливо тільки над вузьким класом програм.

1.2.3. Використання принципів породжуючого програмування для оптимізації

Ідея формально описати необхідну програму і залишити розробити її відповідний вихідний код комп'ютера - завжди інтригувала багатьох дослідників в галузі інженерії програмного забезпечення. Хоча б тому, що породжує підхід забезпечує коректність коду обумовленої побудовою і швидкий (і дешевий) цикл розробки. Нижче буде коротко розглянута методологія, сконцентрована на вирішенні цього завдання, а так само взаємозв'язок з програмною оптимізацією.

Породжуюче програмування - парадигма технології розробки ПЗ, заснована на моделюванні сімейств програмних систем, таких, що по конкретних технічним вимогам можна автоматично отримати спеціалізований і оптимізований проміжний або кінцевий продукт з елементарних, багаторазово використовуваних компонентів реалізації за допомогою бази знань про конфігураціях.

Породжуюче програмування фокусує увагу на сімействах програмних систем, а не на унікальних продуктах. Елементи сімейства не будуються з нуля, вони генеруються на основі загальної породжує моделі (generative domain model), тобто моделі сімейства яка володіє трьома складовими:

- засобами визначення членів сімейства (доменна модель)
- компонентами реалізації (implementation components), з яких може бути зібраний кожен член сімейства - тобто конкретна програма
- базою знань про конфігурації (configuration knowledge). відображає специфікацію для члена сімейства і кінцевого продукту.

Класичною аналогією такого підходу є автомобільна промисловість: є

система з продажу машин, є компоненти,із яких збираються машини і є конфігураційна база знань, визначальна, як збирати автомобілі, відповідно з конкретним замовленням.

Термінологія, описові характеристики, використовувані для визначення членів сімейства, називається простором задачі (problem space), тоді як компоненти реалізації разом з можливими конфігураціями формують простір рішень (solution space).

Рис. 1.2. Діаграма відносин концепцій породжуючого програмування

Дана методологія бачить наступну проблему поточної технології розробки ПЗ: проблема в тому. що в той момент коли ми припиняємо конкретизацію програмної системи, ми не знаємо, як її отримали. Велика частина знань з проектування втрачається, і це робить програмне супроводження складним і дорогим. У породжує програмуванні намагаються відобразити в програмній формі максимум знань про виробництво.

Виробничі знання включають в себе не тільки конфігураційні знання, але також і вимірювальний інструментарій, методики тестування та планування, діагностику помилок, підтримку налагодження, візуальне представлення програми та ін Всі ці різні аспекти відображають всю специфіку предметної

області і упаковані в багато разів використовувані бібліотеки, які називаються *активними бібліотеками*.

Важливою ідей є заміна фіксованих мов програмування на активні бібліотеки мовних абстракцій. Фіксовані мови програмування (наприклад, C + + або java) змушують використовувати певний зафіксований набір мовних абстракцій, в той час як активні бібліотеки дозволяють використовувати набір абстракцій, оптимально сконфігурованих для певної задачі. Вони дозволяють забезпечити справжню мультипарадігменную і предметно орієнтовану під-тримку програмування.

Породжуючі доменні моделі

Породжуюче програмування забезпечує автоматизацію виробництва проміжних і кінцевих продуктів: компонентів і додатків.

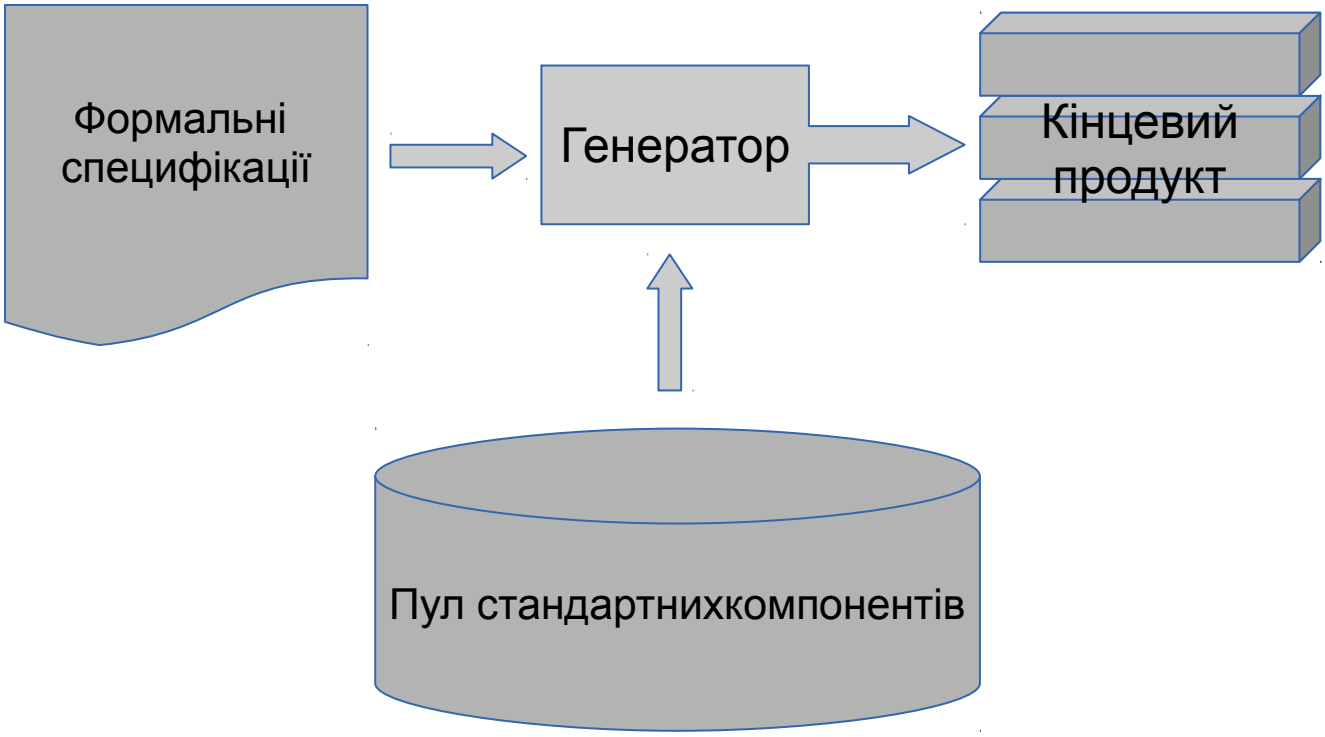

Рис 1.3. Ілюстрація доменної моделі

Необхідними умовами досягнення цієї мети є моделювання сімейств продуктів. Проектування засобів «замовлення» цих продуктів (тобто їх специфікації); забезпечення компонентів реалізації, з яких проводиться збірка продуктів, встановлення відображення від специфікацій продуктів до конкретних збірок компонентів реалізації, а також реалізація цього відображення за допомогою генераторів. «Продуктами», автоматичне виробництво яких можна таким чином організувати, може бути все, що завгодно - від класів або процедур до цілих систем або підсистем.

Архітектура сімейства продуктів необхідна для введення взаємозамінних компонентів ПЗ. Тоді стане можливим швидко і легко визначити - чи відповідає компонент вимогам системи чи ні. Таким чином, необхідні зусилля в бік великої архітектурної стандартизації в різних областях після чого ідея програмних компонентів рушить з місця.

Знання про конфігурації

Знання про конфігурації встановлюють неприпустимі поєднання характеристик (деякі характеристики комбінувати не можна), налаштування за замовчуванням (якщо у додатку не визначені ті чи інші характеристики, вони замінюються допустимими умовчаннями), залежно за замовчуванням (розрахунок деяких «параметрів за замовчуванням» може проводитися з урахуванням інших характеристик), правила конструювання (деякі сполучення характеристик перетворюються на певні поєднання компонентів реалізації) і правила оптимізації (одні поєднання компонентів реалізації можуть виявитися краще за інших).

Знання про конфігурації визначають:

- неможливі комбінації характеристик систем
- налаштування за замовчуванням
- залежності за замовчуванням
- оптимізацію
- конструкторські знання, що фіксують відповідність налаштувань компонентів налаштувань характеристик

Таким чином, в рамках породжуючого програмування необхідно прагнути втілити всі конфігураційні знання у програмній формі.

Простір завдання

У простір завдання входять прикладні поняття і характеристики, за допомогою яких розробники прикладного програмного забезпечення можуть висловлювати свої потреби.

У простір завдання входить декілька типів характеристик.

- Конкретні характеристики. Конкретна характеристика точно відповідає якомусь одному (можливо, параметризованому) компоненту; наприклад, сортування реалізується безпосередньо за допомогою компонента сортування.
- Аспектні характеристики. Аспектах характеристика відповідає аспекту в тому його розумінні, який притаманний аспектноорієнтованному програмуванню. Аспект - це вид модульності, що впливає на безліч інших компонентів; аспектом, зокрема, є декларативне опис синхронізації декількох комнонентів. Аспекти припускають переплітання коду, тобто розміщення коду аспектів в безлічі компонентів.

- Абстрактні характеристики. Абстрактні характеристики не можна реалізувати безпосередньо. Їх реалізація здійснюється через відповідні поєднання компонентів і аспектів. Прикладами абстрактних характеристик є вимоги до робочих характеристик - зокрема, оптимізація, спрямована на досягнення швидкодії, звільнення пам'яті або досягнення точності.

На малюнку 1.4 наведено приклад того, як могли б задаватися характеристики для вибору конкретної бажаної реалізації контейнера (колекції) з набору стандартних:

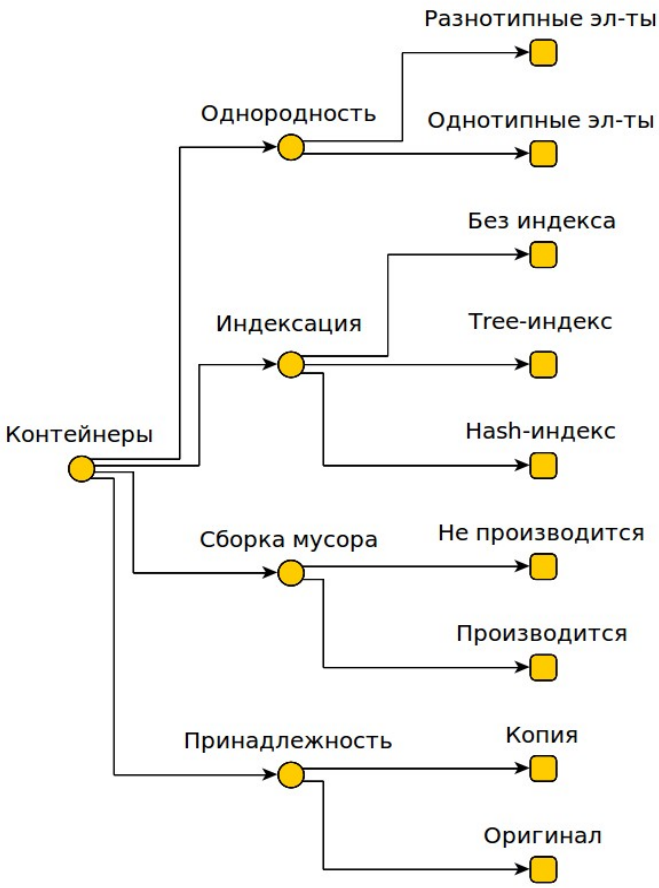

Рис 1.4. Приклад простору завдання для вибору контейнера об'єктів

Особливості проектування в рамках породжуючого програмування

На даний момент найбільш ефективними технологіями для досягнення повторного використання ПЗ вважаються каркаси (frameworks) і компоненти. На жаль жоден з популярних ОО методів аналізу та проектування не підтримує їх розробку в достатній мірі. Для того, що б добитися істотного ступеня повторного використання компонентів слід використовувати прийняту в породжуючого програмування двоетапність циклу розробки:

- один для проектування та реалізації породжуючої доменної моделі. Це розробка «для повторного використання».
- інший - для використання породжуючої моделі при виробництві конкретної системи. Це розробка підсумкових програм з повторним використанням вже готових компонентів.

Обидва процеси відмінні від процесу розробки унікальної системи, такого як, наприклад, уніфікований процес RUP. Мета розробки «для повторного використання» - сімейство систем, а не одиночна система, тому розробка «для повторного використання» повинна бути орієнтованих на надання переваг від повторного використання коштів на систематичній основі.

Варто ще раз підкреслити, що найбільш важливою властивістю циклу розробки «для повторного використання» є орієнтованість на сімейства систем.

При розробці «для повторного використання» виділяють кілька фаз, такі як domain analysis і domain design.

Фаза «domain analysis»

Перший крок полягає у визначенні області дії зацікавленого сімейства систем, тобто у вирішенні питання: які характеристики повинні бути включе-

ні, а які ні. Це вимагає аналізу зацікавлених у проекті осіб та їх цілей поточної і потенційної ситуації на ринку, прогнозування технології і т.д. Дуже важливо правильно визначити область дії сімейства або предметну область, це дозволить уникнути таких узагальнень, де важливі характеристики і змінювані параметри легко втратити, а зайві завести, внаслідок чого може значно підвищитися ціна розробки і супроводу.

Наступний крок у циклі розробки - визначення загальних і змінювальних характеристик членів сімейства і залежностей між змінними характеристиками. Результатом цього аналізу стане документоване використання характеризуючих моделей (feature models), у яких є ряд переваг перед іншими нотаціями моделювання (UML). По-перше, характеристичні моделі явно представляють змінюючи параметри і залежності між ними. Це подання забезпечує базис для виробництва категорій компонентів реалізації, придатних для сімейства систем, засоби для визначення членів сімейства, а також знання о конфігураціях. По-друге, характеристичні моделі розрізняють мінливість в межах члена сімейства і мінливість між різними членами сімейства. Таким чином, уникається впровадження "роздутих" компонентів або каркасів в "роздуті" додатки. Це загальна проблема сучасних компонентних і каркасних технологій, де механізм реалізації мінливості всередині програми (так званий динамічний поліморфізм) також використовується для завдання мінливості між додатками. І нарешті, характеристичні моделі забезпечують незалежні від реалізації засоби представлення мінливості, які дозволяють зберігати рішення про механізми мінливості поза моделлю аналізу. Це і відрізняє дану модель від сучасних ОО нотацій, таких як UML: під час малювання діаграми класів UML ви повинні вирішити, чи треба використовувати спадкування, агрегацію, параметризацію класів або інші механізми реалізації для подання даного змінного параметра.

Характеристична модель може бути представлена у вигляді характеристичної діаграми, т.б у вигляді дерева, де кореневий вузол відображає головні концепти і кожен інший вузол відображає одну з його характеристик. Ребра зв'язуючі вузли, відображають залежності між характеристиками. Нижче наведені різновиди таких залежностей:

- Обов'язкова характеристика
- Опціональна характеристика
- Альтернативна характеристика, тобто з групи подібних характеристик може бути обрана лише одна
- «Або» - характеристка, тобто з групи подібних характеристик можуть бути обрані декілька, або навіть всі, характеристики

Головна перевага таких діаграм в тому, що це дає можливість відкласти визначення того, як саме дана варіативність буде реалізована. Кожна характеристика, кожен вузол - це можливий аспект варіактивності. На цій фазі не потрібно вказувати як дана характеристика повинна бути реалізована, наприклад це може бути як успадкування, так і параметризрвані успадкування або статична параметризація. Фактично, варіативність буде реалізваона порізному на різних рівнях. У підсумку, використання характкерістіческіх діаграм дає можливість проектувальнику аналізувати предметну область без конкретних деталей реалізації.

Фаза «domain design»

Грунтуючись на характеристичних моделях, ми проектуємо породжуючу доменну модель для сімейства, що включає засоби специфікації членів сімейства, загальну архітектуру, яка містить категорії компонентів реалізації, а також знання і конфігурації відображають специфікацію члена в набір компонентів реалізації, які реалізують даний член. У результаті ми реалізуємо

модель, засновану на компонентах і породжують технологіях.

Вся необхідна для цього інформація підготовляється на стадії «domain analysis» і повинна бути закодована в предметноорієнтірованій мові (DSL). Генерація коду діє відповідно з цією специфікацією і виробляє вихідний код компонентів, а так само їх компоновку.

Опис формальної специфікації вимагає певних дій, крім власне складання характеристичної діаграми:

• Визначення головної функціональності на характеристичній діаграмі. Це майбутні категорії компонентів або інтерфейси

• Перерахувати компоненти розділені по категоріях, кожен компонент реалізує загальний інтерфейс категорії.

• Визначити залежності, специфікувати необхідні інтерфейси

• Спроектувати рівні архітектури, всі компоненти можна відсортувати ієрархічно, такий порядок вкаже де є можливість застосувати делегування

Традиційні методи об'єктно-орієнтованого аналізу і проектування, такі як OOSE, OMT і навіть поточна версія RUP, зорієнтовані на розробку одиночних систем і не пристосовані для створення сімейств систем.

Враховуючи, що породжує програмування переслідує саме цю мету, перераховані методи визнаються непридатними для розробки програмних засобів багаторазового застосування, бо для цього потрібно орієнтація на групи систем, а не на окремі системи.

У цьому контексті можна навести такі недоліки традиційних методів об'єктно-орієнтованого аналізу і проектування, що стосуються їх технологічного процесу і нотацій моделювання.

- Відсутність відмінностей між розробкою «для повторного використання» і розробкою з повторний використанням. Для реалізації принципу багаторазового застосування потрібно розбиття процесу розробки об'єктно програмного забезпечення на розробку «для повторного ви-

користання» (тобто інженерію предметної області) і розробку з повторним використанням (прикладну інженерію). Областю розробки «для повторного використання» має бути сімейство систем; лише в цьому випадку по-є можливість виробництва компонентів багаторазового застосування. Процес розробки з повторним використанням повинен бути організований таким чином, щоб у ньому могли бути задіяні повторно використовувані засоби, створені в ході розробки «для повторного використання». Сучасні процеси об'єктно аналізу і проектування не відрізняються такими якостями Ці можна з деякою натяжкою уподібнити лише прикладної інженерії з випадковим (замість систематичного) вживанням повторно використовуваних засобів.

- Відсутність етапу визначення предметної області. Оскільки методи об'єктнооррінтованого аналізу і проектування, орієнтовані на розробку одиночних систем, вони не передбачають визначення предметної області - етапу, на якому проводиться відбір цільової групи систем. Крім того, ці методи спрямовані на задоволення запитів «того самого» покупця одиночній системи; при цьому аналіз кола людей, зацікавлених у появі даної групи систем (і включає потенційних покупців), не проводиться.
- Відсутність відмінностей між моделюванням мінливості в рамках однієї програми і між кількома додатками. У сучасних об'єктноорієнтованих нотаціях не проводиться ніяких відмінностей між мінливістю всередині програми - наприклад, мінливістю об'єктів в часі або застосуванням різних варіантів об'єкта в різних місцях приниження - і мінливістю між додатками, тобто мінливістю, що поширюється на різні програми, призначені для різних користувачів і контекстів застосування. Більше того, ті механізми реалізації, які в об'єктноорієнтованному програмуванні призначені для реалізації мінливості всередині додатків

(наприклад, динамічний поліморфізм), задіюються і для забезпе-чення мінливості між додатками. Наслідком такої практики є поява «роз-дутих» компонентів і каркасів, з яких виходять «роздуті» додатка.

- Відсутність засобів моделювання мінливості, незалежних від реалізації. Крім усього іншого, сучасні об'єктноорієнтован нотації не підтримують незалежне від реалізації моделювання мінливості - наприклад, складаючи діаграму класів UML, доводиться вирішувати, як краще представити

даний змінюючий параметр - шляхом успадкування, агрегації, параметризації класів або за допомогою якого-небудь іншого механізму реалізації.

Простір рішень

Простір рішень, у свою чергу, складається з компонентів реалізації але всіх можливих комбінаціях. Компоненти реалізації розробляються в розрахунку на максимальну сполучуваність (мета домогтися якомога більшої кількості можливих поєднань компонентів), мінімальну надмірність (випадків дублювання коду має бути якомога менше) і граничне збільшення можливостей повторного використання.

При проектуванні простору завдання повинен дотримуватися один важливий принцип; згідно йому, коли ви запитуєте-якої компонент в породжує щей бібліотеці, у вас як у прикладного програміста повинна бути можливість вказати саме той рівень деталізації, який вам потрібен. Ситуація, при якій ви змушені вдаватися до занадто сильною деталізації, неприйнятна в результаті цього ваш клієнтський код стане занадто залежним від реалізації даної бібліотеки. Проте у вас повинна бути можливість вказувати деталі або навіть пропонувати власні реалізації тих чи інших аспектів, якщо в цьому виникне необхідність. Такий ступінь гнучкості стає можливою завдяки наявності

замовчувань, залежностей за замовчуванням і жорстких обмежень (тобто неприпустимих поєднань характеристик).

Поділ на простору завдання і рішень забезпечує можли-ність їх відносно самостійного розвитку. Зокрема, в простір рішень можна вводити нові компоненти або покращувати існуючі і якщо вони при пом продовжують забезпечувати функціональні можливості, що визначаються простором завдання, ніяких змін у клієнтський код вносити буде потрібно. Справа в тому, що клієнтський код замовляє системи та компоненти засобами мови простору завдання; за відображення специфікацій завдання на конфігурації нових компонентів відповідає генератор. Отже, для того щоб впровадити новий компонент, потрібно всього лише внести зміни в генератор. Далі, ми можемо розвивати існуючі предметноорієнтированние мови про-стору завдання і навіть розробляти нові. Цільові компоненти повинні забезпечувати необхідні функціональні можливості, проте абсолютно необов'язково, щоб оптимальна сполучуваність і мінімальна хати-точність були присутні в них із самого початку-з часом їх можна удосконалювати, не порушуючи простір завдання.

Генерація коду

Генератор - це програма, перетворююча високорівневу специфікацію у відповідне більше низькорівневе подання. Прикладами можуть бути просто компілятори, або RMI компілятори, CORBA IDL компілятори і багато інших. Зараз, майже кожен засіб UML моделдірованія або GUI конструктор забезпечують автоматичну кодогенерацію. Концепція генераторів досить популярна, тому що дозволяє:

- Висловлювати в специфікації системи «наміри» проектувальника, справді, описи «намірів» легко розуміти, аналізувати, перевіряти, і під-

тримувати, тобто вони володіють всіми добрими якостями. Вони можуть бути виражені в DSL і оброблені генератором.

- Зміщення верифікації з рівня вихідного коду до рівня специфікацій, т. я. код, вироблений генератором коректний з побудови, і як мінімум частина верифікаційного процесу може бути зміщена на специфікацію, значно зменшуючи складність цієї фази. Для того, щоб цього досягти коректність самих генераторів повинна бути строго доведено методами формального аналізу, це одна з причин високої вартості і складності розробки генераторів.
- Можливість досягнення ефективної реалізації. Генерація результуючого коду зазвичай вельми ефективна. Дополнительно, генератори можуть проводити можливо досить складні проміжні трансформації коду.
- Модулярізація. Так як генератори оперують певною кількістю невеликих, налаштованих компонентів, результуючий код виходить модульним і з низьким ступенем зв'язності, які легко підтримувати, розуміти і повторно використовувати.

Генерація є важливою частиною методології породжуючого програмування. Тут під генерацією розуміється трансляція високорівневої специфікації, вираженої в термінах доменної моделі, або простору завдання у відповідну низькорівневу реалізацію в просторі рішень.

Справді, хоча рефакторінг ПЗ визнається як необхідний етап протягом всього життєвого циклу програми, фактично конкретні приклади вдалого постійного застосування рефакторинга досить рідкісні. Головна причина в тому, що архітектурні, високорівневі характеристики розкидані по всьому програмному коду, тому серйозні зміни зазвичай занадто ускладнені, щоб проводитися. Часто практично неможливо проводити зміни у внутрішньому дизайні ПЗ.

На відміну від цього, при породжує програмуванні, достатньо вносити зміни до високорівневі специфікації, а генератор заново згенерує підсумковий код. Таким чином це дає більше можливостей для ефективного рефакторінга та розвитку.

Висновки

Описаний підхід пропонує інфраструктуру генерації програми з готових компонентів, з метою надання можливості повторного використання вже написаного вихідного коду. Важливим аспектом цього підходу є методика виділення формального опису бажаних характеристик майбутньої програми (формальних специфікацій) і знань про конфігурації, тобто інформації про те, як і які компоненти можна комбінувати, домагаючись бажаного результату. Так як в даній роботі розглядаються оптимізації, пов'язані з взаємодією компонентів, то ймовірний тісний зв'язок між принципами такої оптимізації і принципами породжуючого програмування. І такий зв'язок є: виявилося дуже зручно багато ідей ПП використовувати для, здавалося б, іншої області - для оптимізації. Детально ця тема розкрита в розділі II.

РОЗДІЛ 2. ТЕХНІКА КЕРУВАННЯ ОПТИМІЗАЦІЄЮ ПРОГРАМ

2.1. Загальні принципи

У цій роботі розглядаються техніки, що представляють собою, у відмінність від низькорівневих оптимізацій, зазвичай реалізованих у сучасних компіляторах, більш абстрактний рівень компонентної оптимізації, пов'язаної з агрегуванням і спільною взаємодією різних незалежних компонентів усередині однієї програми.

Як правило, стандартні компоненти написані й підтримуються без врахування того середовища в якому вони працюють у рамках кінцевої клієнтської програми, тому просте їх комбінування зазвичай малоефективне й затратно по швидкодії і пам'яті.

Особливо велику роль відіграє цей недолік у системах автоматично генеруючих код, наприклад, в описаній раніше парадигмі породжуючого програмування(ПП), а також для мов високого рівня, що просто не надають програмісту можливість «ретельно» налаштувати використовувані компоненти.

Зазвичай, при розробці програм, оптимізації такого рівня проводяться вручну програмістом, що, як було показано вище(у розділі 1.1), має ряд недоліків.

У цій роботі, досліджується можливий підхід до розв'язку цієї проблеми, при якому пропонується використовувати автоматичні перетворення, використовуючи додаткову інформацію, впроваджену вручну програмістом у вихідний код у вигляді «анотацій» програми, що описують її властивості.

Виражаючи в такому вигляді додаткові «знання» про компоненти й інші абстракції, що використовуються у програмі, можна досягти виконання предметно-орієнтованих спеціальних оптимізацій властивих конкретній програмі. Крім того, доповнюючи інформацію, вилучену з окремих компонентів, з інформацією про контекст їх використання в клієнтській програмі, про те, як вони один з одним скомпоновані й спільно взаємодіють, дозволяє більш оптимально їх об'єднати й скомпонувати.

Основною ідеєю таких оптимізацій є автоматична модифікація й «сполучення», припасування компонентів взаємодіючих один з одним для досягнення більшої ефективності.

Підставою для цього є інформація про те, які саме компоненти один з одним взаємодіють у даній конкретній програмі.

Оскільки на етапі розробки компонента не відомо в яких умовах він буде працювати, неможливо заздалегідь урахувати й застосувати частину оптимізацій.

Отримана вже на етапі розробки конкретного, «клієнтського» коду (але не раніше!) інформація про те, які компоненти в ньому використовуються, відкриває можливість поліпшити продуктивність даної конкретної комбінації компонентів.

Така можливість є важливим кроком убік розробки програм за допомогою автоматичного вибору й комбінування стандартних компонентів. Інакше кажучи, оптимізації рівня компонентів є досить важливою необхідною умовою для розробки в парадигмі породжуючого програмування (ПП).

Крім того, описані оптимізації є різновидом «спеціалізації» — особливої техніки перетворення програм.

Як було вже описано в розділі 1.2.2, у теорії метаобчислень спеціалізація» використовується для того, щоб оптимізувати код конкретної функції на підставі інформації про заздалегідь відомі значення деяких її параметрів. У

даному трактуванні це «спеціалізація» на підставі інформації про оточення: про те які і як саме компоненти один з одним взаємодіють.

Тобто, такий механізм дає програмісту більш повний контроль над виконанням оптимізацій, доповнює компілятор предметно-орієнтованими оптимізаціями, що властиві конкретній програмі, і дає можливість більш ефективно використовувати сторонні компоненти.

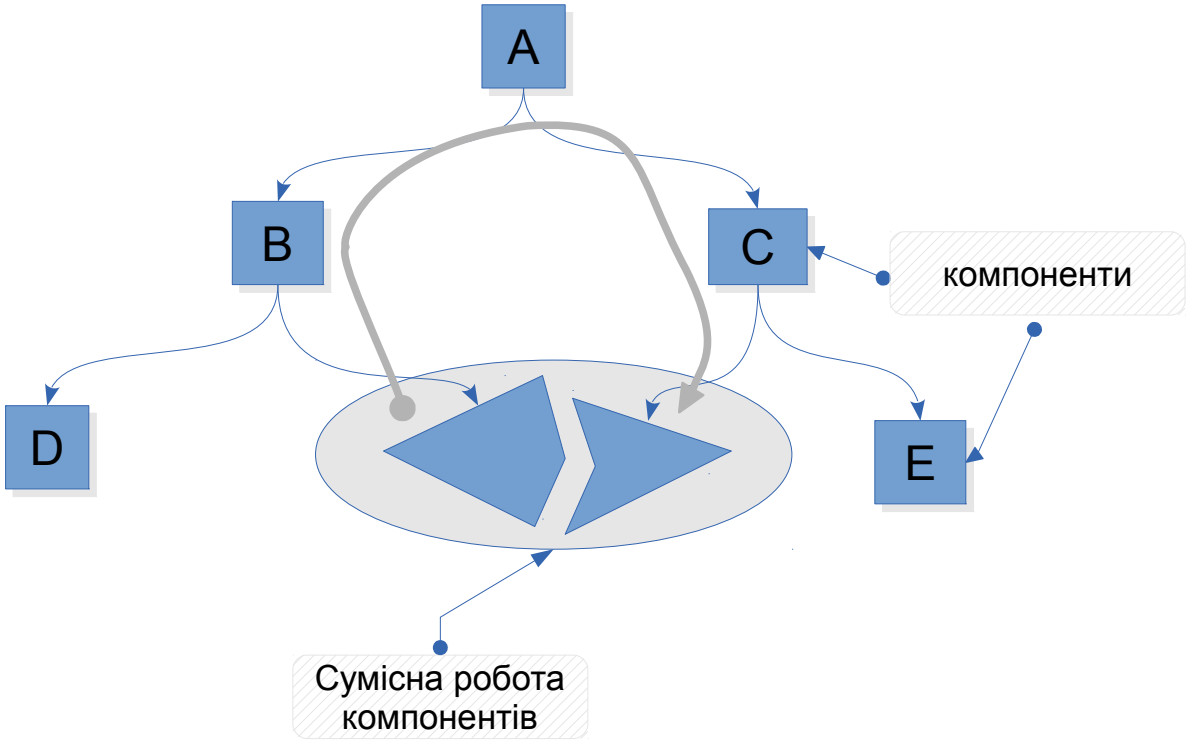

Рис 2.1. Сумісна робота компонентів

На рис 2.1 наведена илюстрація, що пояснює принципі та переваги поєднання компонентів. Сумісна робота компонентів має на увазі спільне використаня даними або іншими ресурсами.

2.2. Анотації у вихідному коді

Анотації — це будь-які метадані, додаткова інформація, що зв'язується з об'єктами використовуваними в програмі, або з ділянками коду, конструкціями й структурними блоками програми.

Взагалі, будь-який компілятор, обробляючи вихідний код створює й використовує «внутрішні» анотації, перевіряючи різні семантичні обмеження та ін. Очевидно, що типізація в мовах програмування сама по собі є заданим фіксованим набором анотацій, що використовуються для статичної перевірки коректного використання об'єктів у програмі. Крім того, анотації широко застосовуються в системах автоматичного доказу коректності програм, тестування і т.п.

Але всім цим широко розповсюдженим прикладам використання властивий цілий ряд недоліків і обмежень:

- Фіксований, жорстко заданий визначений набір анотацій. Неможливість розширення або перевизначення, приводить до того, що не можна описати й відзначити додаткові бажані властивості об'єкта. Якщо розглянути приклад із системою типів, то для багатьох типізованих мов неможливо описати алгебраїчний тип даних (АТД), використовуючи стандартні мовні засоби .
- Неявне використання. Компілятори на різних етапах роботи збирають різні види метаінформації, наприклад для кращої оптимізації, але роблять це приховано й неявно, використовуючи внутрішні структури. Це призводить до неможливості проконтролювати й втрутитися в процес збору й обліку таких анотацій. Це недолік проявляється подвійно, наприклад компілятор «не побачив», не розпізнав можливість застосування деяких оптимизацій там де це можливо (як приклад, відома техніка оптимізації «хвостової рекурсії»)

або навпаки застосував невірну, «поспішну» оптимізацію (як приклад, проблема з невірною елімінацією незмінних змінних у циклі). Крім того, інформація зібрана в одній фазі втрачається в іншій

- Повна автоматизація. Зазвичай метаінформація збирається й обробляється повністю автоматично, наприклад для автоматичного доказу коректності програми. Це тягне проблеми пов'язані з тим, що в складних випадках метаінформацію або не вдається зібрати, або вона невірно визначається. Це може виражатися наприклад у синтаксичних помилках, а скоріше в синтаксичних обмеженнях (наприклад неможливість рекурсивного визначення нового типу).
- Фіксована область застосування. Це означає, що метаінформація використовується лише для деяких, дуже обмежених цілей, наприклад для семантичної перевірки відповідностей змінних і т.п. У програміста немає можливості утилізувати зібрані метадані про програму для якихось інших цілей, не передбачених компілятором або аналізатором. Як приклад можна відзначити, що додаткові семантичні обмеження, можливі при використанні «фантомних» типів або використовуючи «залежні» типи, неможливо описати в менш багатих системах типізації.
- Обмеженість засобів використання. Так наприклад механізм анотацій введений у сучасних промислових ЯП дозволяє зв'язувати анотації з об'єктами, але не з структурними ділянками коду. Крім цього, анотація, що зв'язується з об'єктом не передбачає впливи на зміни інших анотацій.

Тому під «анотаціями» у даній роботі розуміється механізм позбавлений цих недоліків. Саме за допомогою таких перевизначених, розширених анотацій і передбачається досліджувати можливість уведення оптимизацій рі-

вня компонентів. В подальшому, анотації обробляються системою автоматичних розмірковувань, тому запропонований синтаксис схожий на програми мови Пролог: тобто у формі предикатів та функціональних символів. Детальний опис граматики анотацій у BNF-формі наведений у розділі 3.2.

Нижче наведений приклад використання анотації:

```
any:: a->Bool->[a]->Bool
-- saturated(True)
any pred list = foldl  (\res->\x->  res || x) False
                    list
```

Це приклад стандартної функції any, що приймае на вхід список даних, та шукає в ньому елементи, що задовольняють умові pred. При цьому, функція повертає позитивний результат, якщо хоча б один елемент був знайдений, та негативний — в протилежному випадку. Наведений алгоритм неефективний, бо навіть якщо задовільний елемент був знайдений, функція усе одно повинна пробігти по усьому списку.

Виділений середній рядок є анотацією, що передає додаткову інформацію властиву цьому спеціальному випадку поєднанню стандартної функції foldl та функції, що обробляє список елементів: а саме те, що функція отримавши значення True вже его не змінить ні при яких значеннях аргументу, тобто вона насичується(saturated). Тому, ця анотація свідчить о можливості дострокового переривання пошуку після того, як перший задовільний елемент був знайдений.

Якщо анотація не містить назву змінної, то вважається, що вона відноситься до функції біля якої вона визначена. В цьому проявляється сутність анотацій як метаобчислень — вони оперують елементами програми, функціями, деклараціями, змінними тощо.

2.3. Логічні розмірковування використовуючи анотації.

Для того, щоб уникнути зазначених раніше недоліків властивих типовим засобам анотування пропонується додержуватись принципів логічного програмування.

Логічне програмування — це парадигма написання програм для розв'язку завдань, які можуть бути описані в термінах об'єктів (або властивостей об'єктів) і відносин між ними.

Такий вибір цілком природний для анотацій, тому що вони в сутності й описують різні властивості й відносини між програмними об'єктами. Декларативну природу підкреслює той факт, що нас цікавить лише те, які саме анотації пов'язані із програмою, але не спосіб і послідовність їх збору.

У теоретичному плані логічне програмування звичайно зв'язують із формальною логікою, наприклад логікою предикатів або логікою першого порядку. Варто відзначити, що стосовно до анотацій, на практиці виявилось дуже зручно користуватись апаратом модальної логіки, а точніше її різновидом - деонтичною логікою, про що докладно буде описано нижче.

Семантично програми в логічному стилі можна розглядати як опис деякої логічної теорії, тобто як ряд аксіом і теорем у рамках цієї теорії. Виконання таких програм відповідно розглядається як висновок доказу заданих теорем.

Програми в логічній парадигмі складаються з оголошень трьох різних типів:

- Оголошення деяких фактів (або аксіом) про об'єкти й відносинах між ними. Цьому відповідають початкові атомарні відомості, метадані зазначені програмістом.
- Визначення деяких правил про об'єкти й відносин між ними. Цьому відповідають відомості про те, як анотації різних об'єктів зв'язані один з одним, опис властивостей вкладеності, спадкування,

замовчування, правил поширення, та ін. якостей анотацій.

- Формулювання запитів (або теорем, цілей) про об'єкти й відносини. Цьому відповідають твердження, виконуваність яких потрібно перевірити. За результатами таких перевірок можуть виконуватись дії вказані програмістом, що призводить до зміни поведінки або реалізації конкретних компонентів.

Цей підхід не обмежує програміста фіксованим набором можливих відомостей і властивостей, які він може вказати. Йому надається можливість вказувати про програмні конструкції стільки додаткових відомостей, скільки буде необхідно. Керуючи в явному вигляді атомарними відомостями, а також правилами зв'язків між анотаціями, програміст може гнучко втручатися в процес збору й аналізу анотацій, домагаючись бажаного результату. І нарешті, додаючи нові анотації-твердження програміст може виразити додаткові обмеження, що накладаються на семантику різних об'єктів, тим самим довільно розширюючи область застосування анотацій, незалежно від підтримки з боку використовуваного компілятора.

Вертаючись до питання оптимізування, можна помітити, що зазначені властивості анотацій дозволяють застосувати їх для опису нових технік оптимизацій, невідомих компіляторові, а так само для опису ситуацій, коли такі оптимізації можна застосувати (або навпаки, ситуацій, у яких їх заборонено застосовувати).

Не обмежуючись визначеним набором доступних анотацій, програміст може вказувати високорівневі відомості стосовно компонентів або ще більш великих архітектурних блоків, що дає можливість застосовувати оптимізації довільного рівня абстракції.

При розробці деякого компонента, можна врахувати й описати ряд оптимизацій, які спрацюють тільки при виникненні певних ситуацій, у

певних контекстах. Але самі визначення таких контекстів можуть опиратися на інформацію одержану набагато пізніше, вже на етапі використання цього компонента в клієнтській програмі.

2.3.1. Використання ASP модулю прийняття рішень для виконання логічних розмірковувань.

Для збору, аналізу, і автоматичних розмірковувань над анотаціями був обраний модуль прийняття рішень, що підтримує ASP (Answer Set Programming) підхід.

ASP — це особливий різновид декларативного, логічного програмування, зосереджений на знаходженні мінімальної стабільної моделі для набору зазначених фактів/правил. Це пов'язане з теоретико-модельною семантикою (model-theoretic semantics) логічних програм. На практиці це позначає, що ASP вирішувач намагається знайти мінімальний набір значень параметрів повністю задовольняючих усім обмеженням і вимогам логічної програми. Оскільки таких наборів може бути декілька (або нескінченно багато), алгоритми ASP аналізаторів схожі з алгоритмами нелінійного програмування й можуть орієнтуватися на задані фітнес-функції, що вказують ступінь придатності даного конкретного набору параметрів.

Алгоритми ASP забезпечують більш багату підтримку, ніж класичні Пролог-машини, коректно обробляючи ситуації, у яких резолюція пролог-машини заходить у глухий кут, а також підтримує немонотонну логіку, класичне заперечення, замовчування, диз'юнкцію тощо.

2.4. Оптимізації контейнерів даних

У цьому розділі розглянутий приклад високорівневої оптимізації, пов'язаної з автоматичним вибором найбільш прийнятних структур для зберігання даних, використовуваних спільно різними частинами програми.

Кожна програма складається з набору логічно окремих, ізольованих компонентів, взаємодіючих один з одним за допомогою обміну вхідними/вихідними даними, найчастіше спискової природи: різноманітними масивами, колекціями тощо. Для роботи з ними розроблено безліч різновидів контейнерів колекцій, з різними властивостями, характеристиками, обмеженнями й оптимізованими під конкретні способи застосування.

У рамках одного компонента, на етапі розробки є можливість вибрати найбільш прийнятну реалізацію, на підставі інформації про те, як і де використовуються внутрішні дані.

Зовсім інша ситуація при взаємодії зовсім різних компонентів, написаних незалежно один від одного. Оскільки компоненти нічого не знають один про одне, і про те, із чим їм доведеться взаємодіяти, для максимальної спільності, програмісти прагнуть спроектувати їх так, щоб вони отримували на вхід або повертали найбільш прості структури, які легко обробити й в інших компонентах, втрачаючи в такий спосіб перевагу багатьох дуже ефективних спеціалізованих реалізацій.

Автоматичний вибір реалізацій контейнерів, з урахуванням прагнення найбільш повно відповідати специфіці різних компонентів — гарний приклад застосування анотацій.

Насправді, тут буде потрібно, з одного боку, декларативно описати властивості й можливості різних реалізацій (примітивні факти, у термінах логічного програмування), а з іншого — у яких випадках (контекстах) вони можуть застосовуватися (цілі або теореми, у термінах ЛП), а також описати ха-

рактер взаємодії компонент (propagation rules, правила взаємозв'язку між різними об'єктами), і нарешті, особливі обмеження, щоб уникнути вибору некоректних реалізацій.

Крім цього, у багатьох випадках можливі кілька підходящих реалізацій, тому можна продемонструвати застосування різних стратегій, заданих оптимізаційними фітнес-функціями, на підставі яких вибирається найкращий з деякого погляду варіант.

Варто нагадати, що цей природний і очевидний паттерн застосування анотацій близько відповідає опису «знань про конфігурації» з парадигми породжуючого програмування. Так, наприклад, обмеження відповідають «неможливим комбінаціям характеристик систем», анотації вибору найбільш підходящого варіанта відповідають «оптимізації», контексти (опису ситуацій, коли необхідно застосувати ту або іншу реалізацію) — «залежностям за замовчуванням» і т.п.

Нарешті, потрібно відзначити, що всі ці метадані є метаданими «за замовчуванням» не потребуючими втручання програміста-розробника клієнтського коду, що використовує готові компоненти. Але у випадку, коли реальний результат не збігається з бажаним, програміст у будь-який момент може втрутитися в процес, додавши нові анотації: правила, обмеження й описи спеціальних випадків. Усе це можливо опираючись на виразні засоби, підтримувані сучасними ASP вирішувачами.

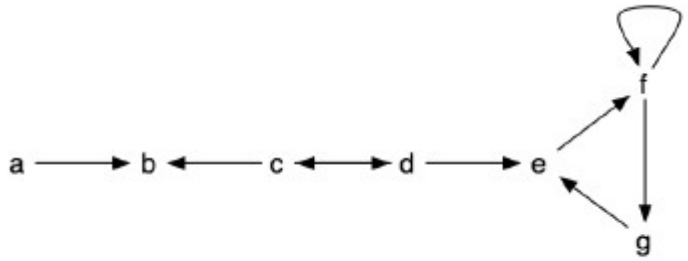

Рис 2.2. Семантика мінімальних моделей

2.4.1. Формалізація властивостей реалізацій контейнерів

Для моделювання були обрані наступні стандартні реалізації контейнерів:

Таблиця 2.1.

Реалізації контейнерів

Назва	Опис	Обмеження
computation	Обчислення списку “на вимогу”	Неефективний пошук заданого ел-ту. Неможливість сортування
Hashset	Структура, що реалізує хеш-словник. Швидкий пошук елемента.	Не підтримує впорядкованість ел-ів
Treelist	Інкапсулює «червоно-чорне» дерево. Швидкий пошук елемента. Підтримка елементів у заданому порядку	
Linkedlist	Зв'язний список. Елементи впорядковані	Неефективний пошук заданого елемента.
Arraylist	Компактний список. Ефективний по вимогах до пам'яті	Неефективний пошук заданого елемента.
Difflist	Компактний список. Зберігає перелік змін ел-ів. Корисний у випадках «майже» константного доступу до ел-ів.	Неефективний пошук заданого елемента. Ефективність падає при великій кількості записів/змін

Для того, щоб коректно вибирати прийнятну реалізацію, необхідно повністю описати мовою анотацій усі переваги й недоліки кожної реалізації, давши тим самим можливість автоматичного висновку за певними правилами.

2.4.2 Формалізація операцій над контейнерами даних

На підставі того, як саме використовуються контейнери, які саме операції над ними проводяться, автоматично вибираються потрібні реалізації. Для того, щоб мати можливість такого вибору, необхідно формально описати відповідність між операціями й відповідними для них реалізаціями.

Таблиця базових операцій, і їх вимоги до реалізації

Таблиця 2.2

Операції над контейнерами

Назва	Опис	Особливості реалізації
at, raccess	Доступ до довільного ел-ту контейнера	Неефективна для зв'язного списку. Для реалізації виду «computation» можливий лише однократний доступ
at, seqaccess	Послідовний доступ до елементів масиву	Припустима для реалізації «computation»
size	Отримання кількості ел-ів контейнеру	Ефективна, якщо контейнер «пам'ятає» свій розмір
map, seqaccess	Перетворення всіх ел-ів контейнеру	Придатна навіть для реалізації «computation»
sort	Сортування ел-ів контейнера	Не підтримується реалізаціями, що не забезпечують порядок проходження ел-ів. Не підтримується реалізацією «computation»
find	Пошук потрібного ел-та	Неефективна для зв'язного списку
insert	Додавання ел-та в довільне місце	Неефективна для компактних реалізацій; Не підтримується для «computation»

Видно, що в різних операціях багато різних умов для ефективного виконання.

Взаємозв'язок між екземплярами контейнерів

У рамках однієї програми звичайно використовується безліч різних контейнерів так чи інакше один з одним зв'язаних. Наприклад, якісь контейнери були отримані сортуванням або іншим перетворенням раніше створених контейнерів. Над кожним з них проводиться свій набір операцій, і відповідно різні екземпляри використовуваних у програмі контейнерів очевидно можуть мати різну реалізацію.

Таким чином, при виборі реалізації для кожного конкретного екземпляру контейнеру необхідно враховувати не тільки вимоги проведених над ним операцій, але й простоту(або складність) перетворення, переносу інформації з урахуванням взаємозв'язку контейнерів між собою. Така інформація відома тільки на етапі написання клієнтського коду (а це означає що вона не може бути передбачена при розробці компонентів) і крім того, у складних, багатоланкових програмних архітектурах вибрати прийнятні реалізації з урахуванням взаємозв'язків — складне завдання навіть для програміста клієнтського коду.

Завдання автоматичного вибору реалізацій у такий спосіб розширюється до того, щоб не тільки вказати для кожного контейнера його прийнятну реалізацію, але й «розмітити» відносини між контейнерами найбільш оптимальним образом. Тут під «відношенням» розуміється спосіб перетворення одного контейнера в інший.

На цю ситуацію можна глянути з іншого боку, з боку проблеми розв'язання конфліктів. Насправді, не завжди вдається підібрати таку реалізацію, яка б задовольняла й підходила під усі вимоги й обмеження програми. Для того, щоб уникнути таких ситуацій потрібно застосовувати т.зв. стратегії розв'язання конфліктів.

Суть таких стратегій у тому, що для різних операцій, у різних

компонентах і т.п., «підсувати» різні реалізації, «безшовно», автоматично їх перетворюючи одну в іншу, без явних вказівок.

Нижче наведений перелік обраних для моделювання стратегій розв'язання конфліктів, вони ж — можливі взаємозв'язки/відносини між різними екземплярами контейнерів.

- «Inherit». Означає, що в породженого контейнера реалізація «успадкована» від батьківського
- «Transform». Означає, що реалізація породженого контейнера перетворена щодо батьківського контейнера, утягуючи при перетворенні й перенесення даних.
- «Multiindex». Описує випадок, коли реалізація породженого контейнера містить у собі додатковий індекс у порівнянні з реалізацією батьківського. Підхід широко застосовується в СУБД, дозволяючи для однієї й тієї ж таблиці/набору даних зберігати кілька індексів. Оскільки при зміні вмісту необхідно перебудовувати й усі індекси, то ефективність використання досягається тільки при read-only доступі.
- Support. Комбінація реалізацій. Контейнер зберігає й «підтримує» відразу кілька реалізацій, використовуючи для кожної операції найбільш прийнятну операцію. Відрізняється від «multiindex» тим, що незалежно зберігає кілька копій даних, а не тільки різні індекси.

У кожного методу зв'язку є переваги й недоліки. Так, наприклад найбільш простий і найменш витратний - «inherit», що означає, що реалізацію змінювати не потрібно, не можна використовувати якщо використовуються операції потребуючі зовсім різних структур зберігання даних. На противагу цьому набагато більш універсальний метод «support», що неявно підтримує

стільки реалізацій, скільки буде потрібно, дуже неефективний по пам'яті — припускаючи зберігати кількох копій даних, і в той же час неефективний по продуктивності — при кожній зміні даних потрібно вносити корективи в кожну «підтримувану» структуру.

У якості короткого прикладу, що пояснює вищеописане, можна навести наступний рядок коду:

b = sort(a), де a, b — це спискові контейнери.

Одним з можливих результатів розмітки такого коду, може бути наступна інформація:

«a»: Hashset,

«a'»: Linkedlist,

«b»: Linkedlist,

«a — a'»: transform,

«a' — b»: inherit.

Тут, «a'» - це проміжна змінна, «a — a'», «a' — b» - це типи відносин між змінними. Такий розв'язок позначає, що спершу створюється проміжна змінна «a'» з відмінною від «a» реалізацією, у неї копіюються дані, і потім її вміст сортується зі збереженням результату в змінній «b» використовуючи таку ж реалізацію.

Таким чином на підставі зв’язків мж екземплярами контейнерів та операціями, що над ними проводяться можливо підібрати найбильш прийнятні реалізцації

2.4.3. Фітнес-функції: вибір найкращого розв'язку

Майже завжди можливо підібрати декілька «мінімальних моделей» (у термінах ASP) — варіантів розв'язку, варіантів анотування обмеженням, що задовольняють усім особливостям як використовуваних операцій, так і використовуваних реалізацій контейнерів.

Як приклад, можна привести «Treelist» - це найбільш універсальна реалізація, яка з однієї сторони дозволяє проводити досить ефективний пошук потрібного ел-та (логарифмічна складність), а з іншого — підтримує порядок проходження ел-ів (звичайно, тільки для ел-ів утворюючих решітку). І в більшості випадків «Treelist» задовольняє всім вимогам програми, тому можна (майже) завжди використовувати саме цю реалізацію. У цьому виродженому випадку губились б переваги використання більш спеціалізованих і ефективних реалізацій.

Таким чином, зрозуміло, що встає проблема вибору найбільш вірного варіанта анотування, який не тільки задовольняє всім вимогам і обмеженням, але й робить це якнайкраще. На щастя,ASP алгоритми прийняття рішень, на відміну від більш традиційних Пролог-машин вміють справлятися з такими ситуаціями, використовуючи крім логічної програми ще й фітнес-функції, що обчислюють «оцінку» — числовий еквівалент корисності й ефективності кожного варіанта розв'язку. Процес вибору найбільш кращого розв'язку — є пошуком екстремуму фітнес-функції, її максимізації.

Детальне пророблення фітнес-функцій є окремою великою темою. Тому в даній роботі її вигляд дещо спрощений.

Семантика фітнес-функції

Фітнес-функція побудована з урахуванням прагнення оптимізувати наступні показники:

- Введений захід ефективності виконання операції. Так, для кожної пари («Операція», «Реалізація») — проставлена оцінка, що показує ефективність із якою може бути виконана «Операція» в даній «Реалізації». Оцінка проставлена відповідно до класу «Big-O» нотації складності виконання. 0 — логарифмічну складність виконання, 1 — константну складність. Сумарна оцінка заходу ефективності в обсязі всієї програми повинна бути найбільшою.
- Введені штрафи за використання неефективних структур даних. Наприклад пошук ел-та у зв'язному списку — це неефективна операція, що має лінійну складність. За кожну неефективну операцію нараховується штраф. Очевидно, що сумарний штраф в обсязі всієї програми повинен бути мінімальним.
- Ранжирувані методи перетворення контейнерів. Сумарна кількість перетворень повинна бути мінімальною.
- Облік абстрактних характеристик. Цей термін відноситься до термінології породжуючого програмування, і в даному конкретному випадку означає облік деяких властивих реалізаціям характеристик. Наприклад, віддається перевага компактним структурам даних, через їх більш ефективного використання пам'яті.
- Установлені додаткові обмеження. Такі обмеження вказують на варіанти розв'язку які досить погані, щоб їх відкинути ще до етапу підрахунку оцінок. Наприклад, використання неефективних операцій у циклі.

2.4.4. Алгоритм оптимізації контейнерів даних

Витягуючи метаінформацію, анотації з вихідного коду стандартних компонентів системи формують, виражаючись термінами експертних систем, експертні знання, базу знань. Ця база складається з анотацій, що описують характеристики реалізацій, вимоги операцій, залежності й параметри за замовчуванням, правила розв'язання конфліктів або правила взаємозв'язків, додаткові обмеження, що описують неможливі розв'язки, і нарешті, оцінки розв'язків і опис фітнес-функцій.

З іншого боку, аналізуючи вихідний код клієнтської програми, формується оперативні, «тактичні» знання, факти, застосовні тільки для цієї програми. Сюди входить дерево екземплярів контейнерів і додаткові анотації, уведені програмістом клієнтського коду, що перевизначають або уточнюють частину раніше зібраних анотацій. Важливо, що ці анотації використовуються «ad-hoc» і не можуть вплинути на обробку інших програм.

Дерево екземплярів — це дерево, вузлами яких є всі використовувані в програмі змінні що містять екземпляри контейнерів. А ребра між вузлами — це залежності між змінними-екземплярами. Вони відбивають порядок створення контейнерів, шляхом копіювання або перетворення один в одного.

Кожному вузлу приписуються операції, які фактично в програмі обробляють відповідний до цього вузла екземпляр (змінну). А кожному ребру приписується вид взаємозв'язку (усі можливі види були описані вище).

На наступному етапі перевіряється відповідність отриманого дерева всім обмеженням.

Для цього виділяються кластери. Кластер — це набір суміжних вузлів, що мають між собою ребра типу «inherit». Інакше кажучи, кластер — це набір змінних, що мають однакову реалізацію. Коренем такого кластера є

вузол-змінна, з якого прямо або побічно утворені всі інші вузли кластера. Це позначає, що всі змінні «успадковують» реалізацію кореня кластера. Реалізація кореня кластера вибирається так, щоб всі операції приписані іншим членам кластера «дозволяли» її використання.

Якщо таку реалізацію знайти неможливо, кластер ділиться на більш дрібні суб-кластери, намагаючись вже для них ізольовано знайти відповідну реалізацію. Зв'язки між кластерами — це ті або інші перетворення: зміни реалізації, додавання нового індексу й т.п.

Оптимізація фітнес-функції прагне розбити дерево на як можна меншу кількість кластерів (тобто використовувати якнайменше різних реалізацій), але в те ж саме час, щоб реалізація обрана для кожного кластера добре справлялася з операціями, з ним зв'язаними.

Ще раз варто повторити, що більша частина анотацій — статичні, стратегічні знання, прописуються однократно на етапі розробки конкретного компонента. Із клієнтського коду виділяється лише інформація о використанні компонентів, за винятком випадків, коли програмістові клієнтського коду дійсно потрібно щось виправити й налаштукати.

Таким чином, додаткова інформація, включена в код компонентів, «включається» і «спрацьовує» лише в з'єднанні з оперативною, тактичною інформацією про графа керування в програмі, або про графа використання.

Приклад з вибором реалізації контейнерів демонструє застосування саме графа/дерева використання цих компонентів.

Інформація для автоамтичних розмірковувань збирається як на підставі вручну внесених анотацій, так і на підставі синтаксичного аналізу, аналізу потоку даних тощо.

Автоматичне перетворення коду

Перетворення програмного коду, вручну або автоматично, використовується для різних цілей і відповідно проводиться різними способами. Розглянемо докладніше взаємовідношення між двома різновидами перетворення: оптимізацією й рефакторингом.

Оптимізація — це процес еквівалентної зміни коду програми з метою прискорити продуктивність. «Агресивні» техніки змінюють код до невпізнанності, порушуючи інкапсуляцію й логічні зв'язки між програмними об'єктами. У той же час, рефакторинг — процес еквівалентної зміни з метою полегшення розуміння тексту програми, пошуку помилок у ній, спільної розробки командою розробників, і так само, як наслідок, полегшення супроводу й підтримки. Проводиться рефакторинг виділенням логічно ізольованих частин, домагаючись створення ясної структури програми, а так само внесенням конструкцій, що пояснюють намір програміста, що обмежують, що й пропонують певну поведінку й взаємодію об'єктів.

Як вже було показано, рефакторинг і оптимізація в загальному випадку використовують протилежні один одному стратегії й часто доводиться вибирати котрій з них віддати перевагу, і в якій мірі.

Для того, що б послабити в деякій мері це протиріччя можна використовувати ряд автоматичних стратегій перетворюючих «оригінальний» зручний для розробки код у його більш оптимізовану версію на етапі безпосередньо компіляції.

Нижче будуть розглянуто кілька таких стратегій або ідіом з погляду здійсненності на основі використання анотацій.

Ітератори.

Ітератори — це ідіома, заснована на ідеї програмно виразити спосіб використання деякого об’єкта клієнтським кодом. Таким чином, програміст,

надаючи інформацію про те, як він використовує даний об’єкт, дозволяє це врахувати й згенерувати більш ефективний код для даного конкретного способу використання.

Ітератори в більшості випадків використовуються у зв'язку з колекціями. Ітератори послідовного доступу, довільного й т.п. — інкапсулюють стратегію обходу елементів колекції. Від необхідності створювати і явно використовувати ітератори програміста рятує той факт, що можна автоматично визначати засіб обходу колекції й автоматично перетворювати код, пристосовуючи його саме до використовуваного типу обходу.

Для функціональних мов класичним способом обходу колекцій є мапери. Так, для мови haskell визначена функція вищого порядку map у такий спосіб: map :: (a -> b) -> [a] -> [b], що означає, що вона приймає на вхід список ел-ів і функцію перетворення, і повертає новий список, що складається з перетворених елементів старого списку. Функція перетворення — це аналог тіла циклу для імперативних мов, одержує як параметр черговий елемент вихідного списку, а результат цієї функції — стає черговим елементом підсумкового списку. Оскільки функція перетворення може використовуватись в різних контекстах, у загальному випадку усередині неї незрозуміло в якій послідовності вибираються й подаються на її вхід ел-ти. Недолік інформації про контекст є зворотною стороною інкапсуляції й абстрагування коду перебору ел-ів від коду перетворення конкретного значення.

Інформацію про контекст використання можна заповнити, використовуючи анотації. Тоді, усі звертання до ел-ів списку усередині даного циклу позначаються анотацією контексту, що дозволяє вгадати до якого ел-ту відбудеться звернення наступного разу, і, наприклад, заздалегідь його закешувати.

Flyweight

Ця ідіома дозволяє зменшити обсяг пам'яті що зайнята колекцією об'єктів, уникаючи дублювання однакових частин цих об'єктів. Її застосування можна розглядати як засіб зберігання колекції у вигляді стислого, упакованого представлення колекції в пам'яті. Ця ідіома звичайно реалізується створенням особливого об'єкта-«фабрики», що формує об’єкт-член колекції «по запиту», як би «розпаковуючи» його дані, його внутрішній стан зі стислого представлення.

Недоліком використання цього паттерна є зниження наочності вихідного коду програми, у якій (якому) замість явно зазначеної колекції об'єктів фігурують «фабрики», приховуючи інформацію про тип оброблюваних даних. Крім того, використання цього паттерна зменшує ступінь повторного використання коду, не даючи можливості передавати дані на обробку компонентам і функціям, що ухвалюють саме колекції в явному вигляді, наприклад, немає можливості використовувати стандартні засоби для обробки колекцій у такому випадку. Зазначені недоліки перешкоджають широкому використанню цієї корисної ідіоми.

Застосування анотацій може дозволити автоматично застосовувати цей паттерн у випадках коли це необхідно. Програміст описує за допомогою анотацій лише ті дані (поля), які не повинні дублюватися.

Кешування

Одним з основних способів оптимізації практично будь-якої комп'ютерної програми є кешування. Кешування — це виділення й збереження проміжних даних для повторного використання тим самим або іншими компонентами.

Зазвичай, в імперативних мовах для кешування використовують рі-

зновиди паттерна Registry — тобто створення окремого об’єкта-«одинака», який зберігає додаткову інформацію, доступну для читання/запису по ключу. Недолік цього підходу в складності відстеження, що інформація кешується раніше, ніж вона використовується, у складності відстеження актуальності збереженої інформації й т.п.

У середовищі функціональних мов такий підхід не використовується через руйнуючу семантику цього паттерна, що веде до побічних ефектів. Тому використовується ряд стратегій на основі використання кортежів (tupling). Суть підходу в тому, щоб перетворити функцію так, щоб вона крім основного результату повертала також результати проміжних обчислень, які передаються як аргументи наступної функції.

Складність автоматичного застосування цієї стратегії в тому, щоб розпізнати загальні підвирази в різних функціях. Цю проблему можна виправити використовуючи анотації для того щоб дані, доступні для повторного використання, позначати тегами, тим самим указуючи системі які саме частини використовуваних підвиразів поєднувати з результатом і передавати як параметри для інших функцій.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ЗАСОБУ КЕРУВАННЯ ОПТИМІЗАЦІЄЮ ПРОГРАМ

Керування оптимізацією програм за допомогою анотацій виконується окремою фазою трансляції вихідного коду на мові програмування Haskell, що також містить додаткові анотації, у байт-код віртуальної машини LLVM.

Тому, для демонстрації, було побудовано консольну утиліту, написану на мові програмуваня C++, яка працює у UNIX середовищі, та являє собою транслятор, що містить у собі цю специфічну фазу.

Ця утиліта приймає на вхід текст комп'ютерної програми, проводить компіляцію, залучаючи фазу виконання компонентно-орієнтованих оптимізацій, генерує байт-код віртуальної машини LLVM, та виконує його, використовуючи штатний JIT-компілятор віртуальної машини LLVM.

Окрім цього, на фазі обробки анотацій залучається функціонал сторонньої утіліти clasp — системи автоматичних міркувань у стилі логічного програмування ASP. Формується особлива логічна програма, що дотримується синтаксису clasp, та передається для виконання автоматичних міркувань. Результати знайдених розв’язків використовуються для точного визначення усіх необхідних параметрів автоматичного перетворення вихідного коду програми задля оптимізації.

Загальна структура демонстраційної утіліти пояснюється у розділі 3.1. Деталі обробки анотацій та стислий опис системи clasp надано у розділі 3.4. Деталі генерації байт-кода та стислий опис віртуальної машини LLVM надано у розділі 3.5.

Ефекти виконання запропонованих оптимізацій аналізуються на прикладі декількох нескладних тестових програм.

3.1. Загальна структура транслятору

На рис. 3.1. приведена загальна структура розробленого транслятору.

З лівого боку зображені модулі, що є типовими для компіляторів. Це є лексичний та сннтаксичний аналізатори, на виході з яких формується AST — абстрактне синтаксичне дерево. Грунтуючись на цьому дереві наступний модуль — семантичний аналізатор – будує структури, що пригодні для наступних трансформацій та трансляції.

З правого боку зображені модулі, що обслуговують обробку анотацій. Так, екстрактор анотацій збирає і відокремлює від віхидного коду програми усі знайдені анотації. Знайдені анотації за допомогою результатів статичного аналізу, виконаного семантичним аналізатором перетворюються аналізатором анотацій до особливої логічної програми, придатної до обробки стороннюю системою проведення автоматичних міркувань.

Керуючись розв’язками логічної програми окремий модуль виконує відповідні еквівалентні перетворення програми на рівні проміжного представлення вихідного коду.

Отримане після усіх наданих трансформацій остаточне проміжне представлення транслюється генератором байт-коду у програму на мові IL LLVM, що розміщується безпосередньо у пам’яті. Надалі ця програма виконується штатним JIT-компілятором віртуальної машини LLVM.

Усі проміжні шаги виконуються беспосередньо у пам’яти.

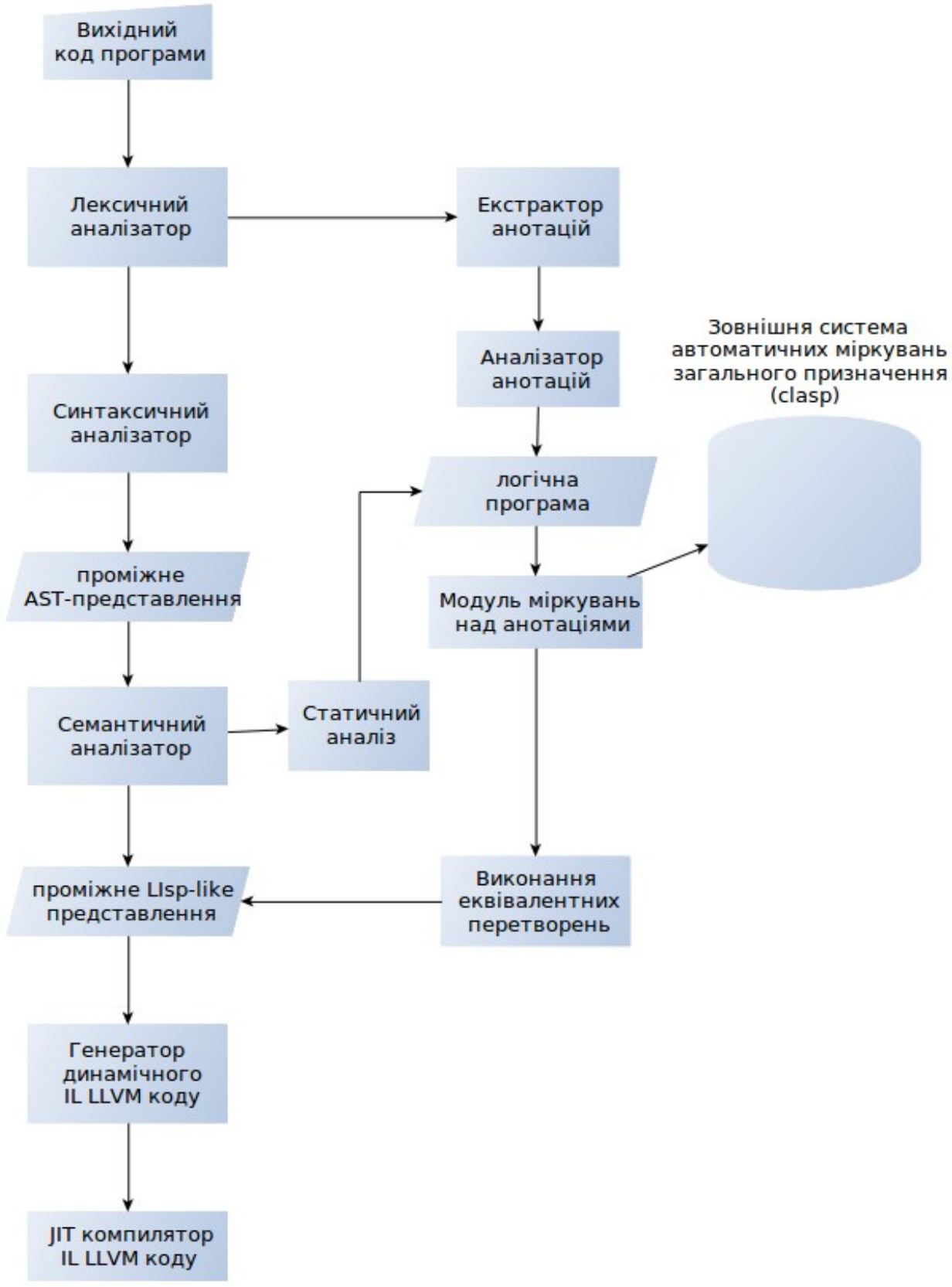

Рис 3.1. . Структурна схема транслятору

3.2. Опис лексичного та синтаксичного аналізаторів

Лексичний те синтаксичний аналізатори були автоматично сгенеровані на підставі формальної специфікації у форматі ANTLR 4. Таким чином було реалізовано распізнавання підмножини сіитаксису язика программування Haskell.

Граматика підмножини Haskell відображена у додатку 1. Нижче наведена формальна граматика для відокремлення анотацій від основного вихідного коду програми та іх подальша обробка.

```
Predicate ::=  predname
        | predname(Expr1,..,Exprk)
        | predname(metavar, Expr1, ..., Exprk)
Expr    ::=  empty   |   var |   num
                     |   literalconst
                     |   Fun
Fun     ::= funname (Expr1, ..., Exprk)
            |  funname(metavar,  Expr1, ...,
Exprk)
```

Лексичний аналізатор відокремлює коментарі з вихідного коду програми та передає їх модулю-екстрактору анотацій. Увесь інший текст програми передається синтаксичному аналізатору.

Мета роботи лексичного та синтаскичного аналізаторів побудувати AST — спеціальну структур даних, що відображае синтаксичну структуру программи.

На рисунку 3.2 відображена UML-діграма базових класів лексичного та синтаксичного аналізаторів. Клас HkGrammarLexer оброблює вхідний набір символів, та, проводячи лексичний аналіз, заповнює структуру TokenStream,

що инкапсулює набір роспізнаних токенів. Цей набір є вхідним до класу HkGrammarParser, код якого автоматично сгенерований за допомогою генератора системи ANTLR v3.

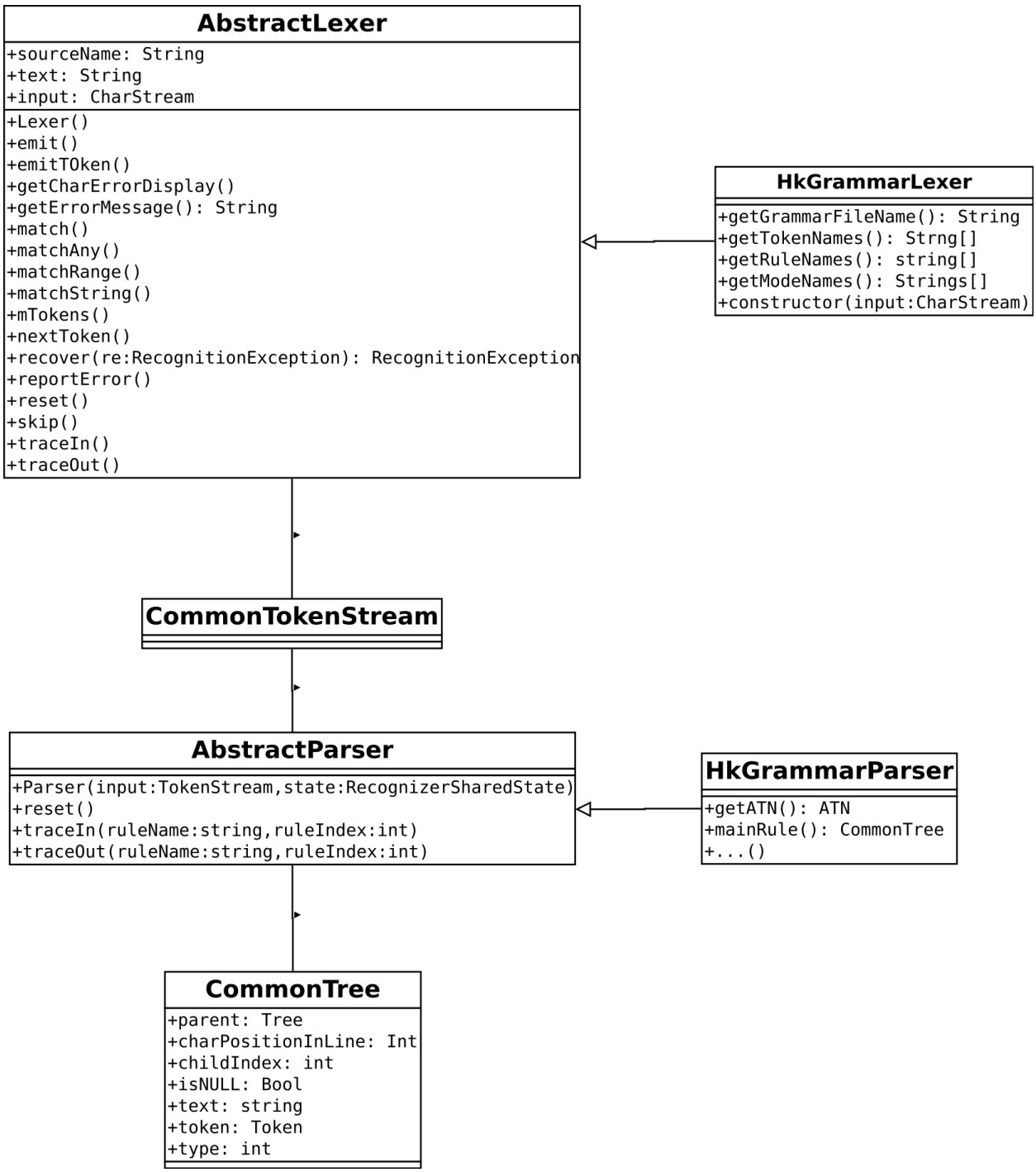

Рис 3.2. UML діаграма модулів лексичного та синтаксичного аналізаторів

Клас HkGrammarParser инкапсулює синтаксичний аналізатор, що на основі даних з TokenStream будує абстрактне синтаксичне дерево.

Абстрактне синтаксичне дерево відображає синтаксичну структуру програми, та відповідає класу CommonTree — це древовидна структура, кожен вузол якої містить відповідний лексичний токен з вихідного коду програми.

На рис. 3.3 наведений приклад абстрактного синтаксичного дерева для наступного коду:

```
inc:: Int->Int
inc x = x + 1Рис 3.3.
```

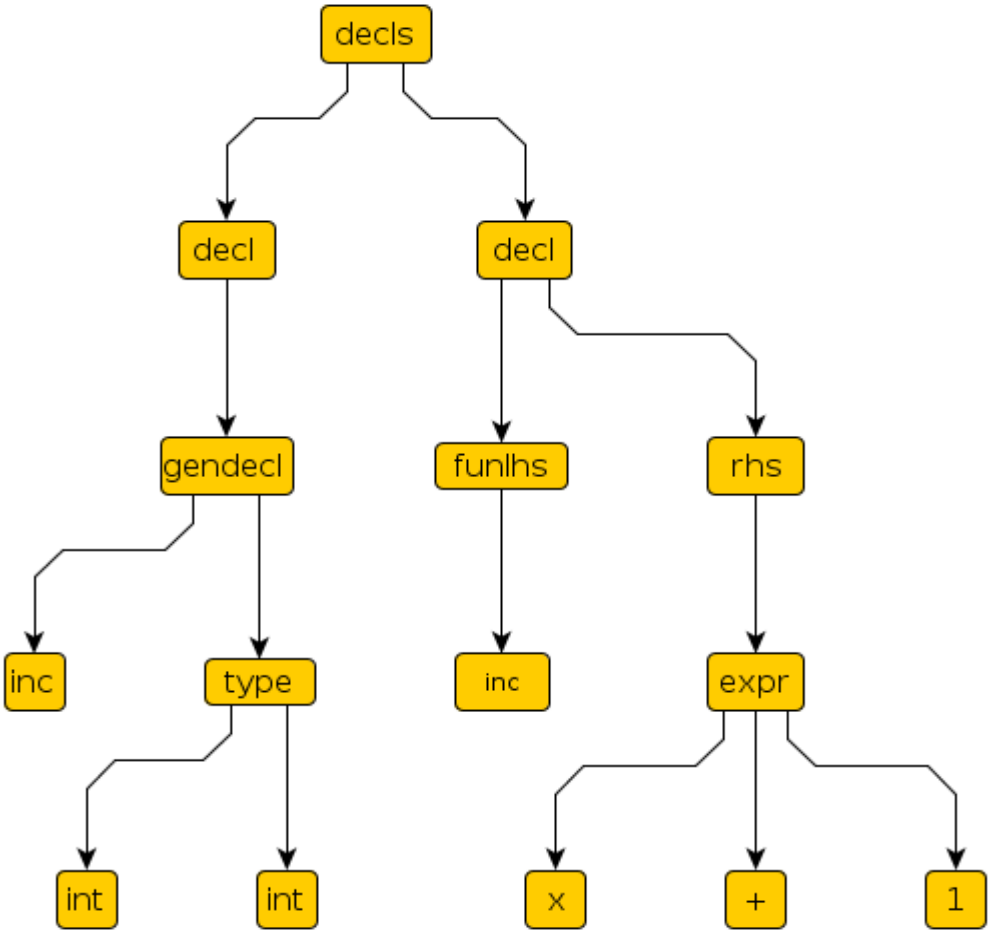

Рис 3.3 Приклад AST дерева

3.3. Семантичний аналізатор

Семантичний аналізатор обробляє AST дерево, отримане на попедньому етапі. В результаті такої обробки перевіряється семантична коректність програми, заповнюються внутрішні структури с інформацією о програмі.

С точки зору анотацій важлива таблиця вибору реалізації функції грунтуючись на типах змінних.

Наприклад:

```
class List s a where
    size':: a->Int

instance List ComputationList [a] where
    size' list = length list
instance List ArtrayList [a] where
    size' list = 1 + length list
```

Цей код дозволяє використовувати одну з декількох реалізацій функції size` тому потрібно для кожного випадку використовування визначити на підставі информації о типах змінних потрібну реалізацію. Таким чином, будується таблиця підтримки “ad-hoc” поліморфізму.

Нижче неведений приклад такої таблиці:

Таблиця 3.1.

Приклад таблиці підтримки поліморфізму

ідентификатор	**Назва ф-ції**	**Реалізація**
1	sort	1
2	sort	1
3	sort	2

На прикладі видно що в деякій програмі, різні визови ф-ції sort виконуються різними реалізаціями.

Ця структура даних важлива тим, що саме на неї впливає результат логічних розмірковувань над анотаціями. Тобто наразі, вплим анотацій на обработку програми обмежений маніпуляцію с даними цієї таблиці, що дозволяє використовувати в кожному окремому випадку саме потрібну реалізацію.

Результатом семантичного аналізу є структура у вигляді дерева, під назвою S-expressions, тобто оберенена польска нотація.

На рис 3.4 приведений приклад цієї структури для програми виду:

```
main = func1 (a + b, 8)
```

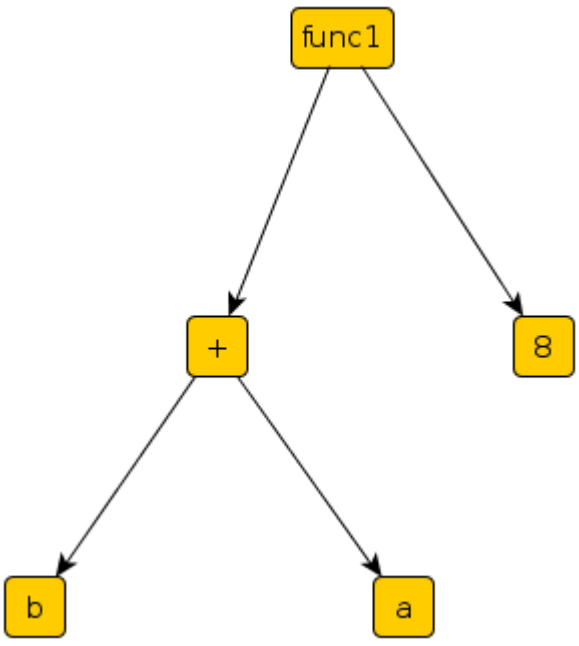

Рис 3.4. Приклад S-expressions дерева

3.4. Обробка анотацій

На фазі лексичного аналізатору анотації відокремлюються та обробляються окремим чином. Наприклад у коді наведеном нижче, лексеми, що знаходяться у коментарях збираються та подаються на вхід екстрактора анотацій.

```
any:: a->Bool->[a]->Bool
-- saturated(True)
any pred list = foldl  (\res->\x->  res || x) False
                    list
```

Екстрактор анотацій відокремлю коректні анотацій та підготовлює їх у форми, прийнятноі до обробки аналізатором анотацій. Аналізатор анотацій користується декількома джереами анотацій збираючи их усі у купу. На рис 3.5.наведена диаграма джерел анотацій.

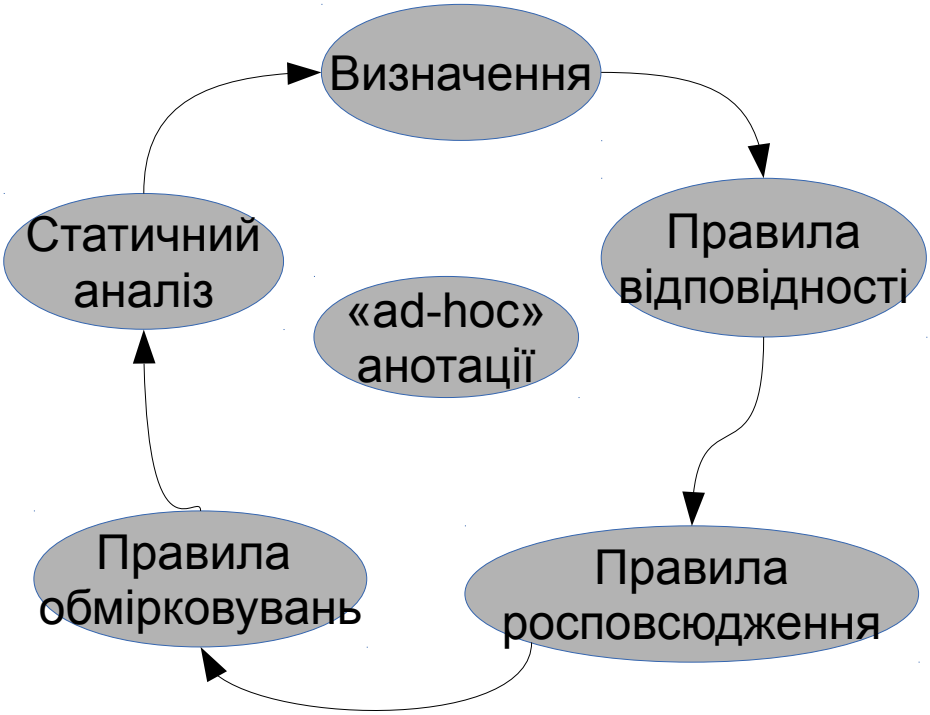

Рис 3.5. Різні види анотацій

Ці іиди анотацій збираются з: окремих файлів, коду біблиотек и

компонентів, коду клієнтської програми. Крім того ці штучно сформовані анотації доповнюються даними статичного аналізу та виводяться на підставі правил розмірковувань.

Рис 3.6. Процес сбору та обробки анотацій

Таким чином, анотації збираются и подаються у вигляді логічної програми на вхід сторонньої утіліти, що виконує автоматичні розмірковування. В даній реалізації розв'язки, шо були знайден, впливають на таблицю підтримки поліморфізму, тим самим це дає можливість обрати потрібну реалізацію для кожної програмної ф-ції.

3.5. Демонстрація використання

3.5.1. Демонстрація відкладених обчислень

Як було вже показано серед змодельованих реалізацій контейнерів є така, що пов'язана з відкладеними обчисленнями. Особлива відмінність цієї реалізації полягає в тому, що замість значень елементів, в цьому контейнері зберігається алгоритм(функція), що генерує значення елементів при потребі. Цей розділ призначений щоб продемонструвати використання цього контейнеру.

Для цього, розглянемо наступну функцію:

```
example:: Int->Int
example x = sum $ filter (>x) [1..10]
```

Ця фунція розраховує суму елементів у списку, що більше за даний. Щоб явно відобразити проміжні змінні іі потрібно переписати наступним чином:

```
example:: Int->Int
example x = let a = [1..10],
                   b = filter (>x) a
              in sum b
```

Наступним кроком потрібно дати означення стандартним функціям `sum` і `filter`:

```
sum :: [Int]->Int
sum list = foldl (\acc->(\el-> acc + el)) 0 list
filter :: (Int->Bool)->[Int]->[Int]
```

```
    filter ffn list = foldl (\prefix-> (\el->
    if (ffn el) then (prefix ++ [el]) else prefix)) []
list
```

Таким чином, враховучи, що параметри передаються за значенням, програма містить 4 проміжні змінні:

```
    a = [1..10]
    tmp2 = a /внутрішня змінна filter/
    b = filter .. a
    tmp3 = b /внутрішня змінна sum/
```

На рисунку 4.1 відображені 4 проміжних контейнера даних у вигляді кіл. Стрілками позначена операція копіювання вмісту контейнерів. Також показано, над якими контейнерами які операції виконуються. Зокрема “seqaccess” позначає операцію що потребує послідовного доступу не змінюючи змісту контейнера. Еліпсом позначені ті проміжні змінні, що відносяться до одного кластеру, тобто повинні мати однакову для всього кластеру реалізацію. Це є наперед задане обмеження, виходячи з того факту, що змінна “b” є результатом операції фільтрування над проміжною змінною “tmp2”.

Таким чином, задача системи прийняття рішень полягає в тому, щоб на підставі інформації о том, які операції проводяться над контейнерами, враховуючи вказані обмеження, вибрати найбільш оптимальні реалізації для кожного контейнеру, в той же час, намагаючись мінімізувати кількість перетворень при копіюванні даних.

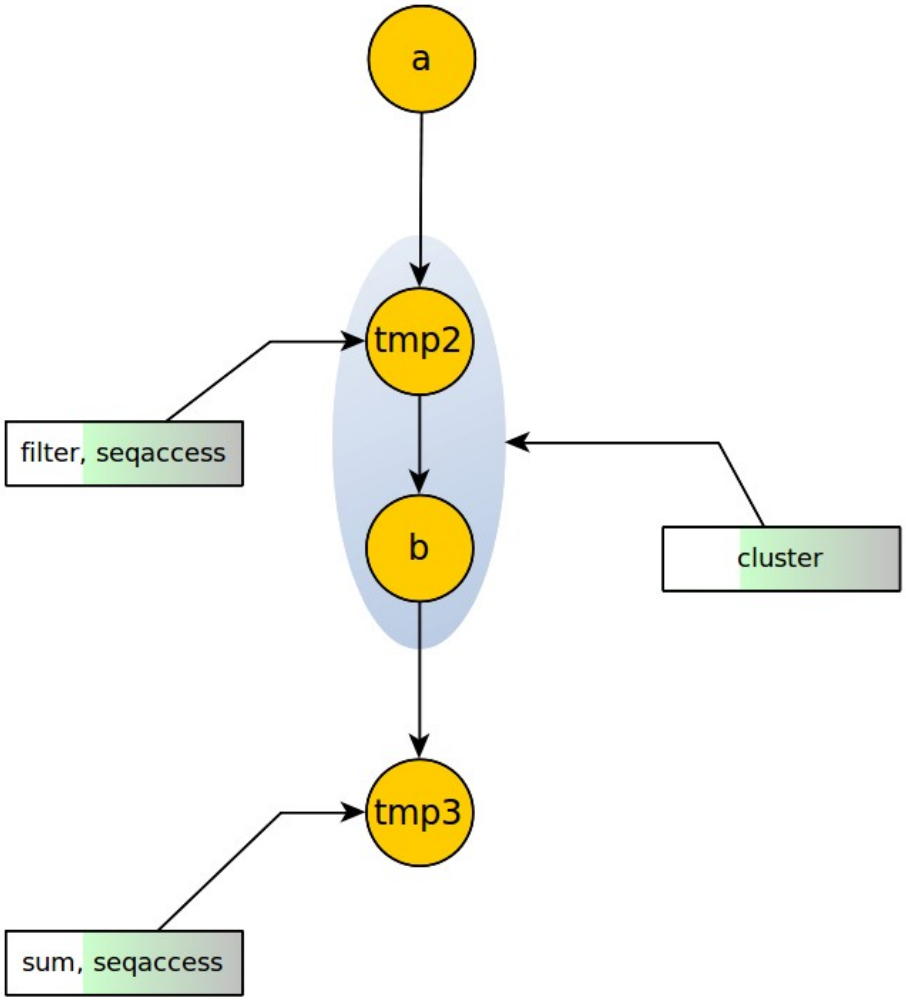

рис 4.1. граф потоку даних

Частина логічної програми, що пов’язана с результатами статичного аналізу DFA — тобто аналізу потоку даних, граф якого приведений на рис. 4.1. виглядає наступним чином:

```
var(a).var(tmp2). var(b). var(tmp3).
var_connection(a, tmp2).
bind_var_op(tmp2, seqaccess).
var_connection(tmp2, b, ct_inherits).
```

```
var_connection(b, tmp3).
bind_var_op(tmp3, seqaccess).
```

Предикат `bind_var_op(var, op)` ставить у відповідність контейнеру `var` операцію `op`, що над ним виконується. Предикат `var_connection(var1, var2, type)` ставить у відповідність тип зв'язку між контейнерами.

Система прийняття рішень находить наступну оптимальну стратегію:

```
bind_var_impl(a, computation).
var_cluster(a, a).
var_cluster(a, tmp2).
var_cluster(a, b).
var_cluster(a, tmp3).
```

Це рішення позначає, що усі змінні входять в один кластер – тобто мають однакову реалізаціє, а саме реалізацію "computation", тобто програма не потребує додаткової пам'яті для зберігання проміжних контейнерів.

Висновки

Цей приклад продемонстрував, що система прийняття рішень коректно розпізнає можливість використовування відкладених обчислень, а також надає програмисту інструменти важілі впливання на співвідношения швидкість/об'єм пам'ті, використовуючи відповідні анотації.

ВИСНОВКИ

ДОДАТОК А. РЕАЛІЗОВАНА ПІДМНОЖИНА СИНТАКСИСУ HASKELL У НОТАЦІЇ BNF

```
atomic    ::=   literal | variable  |   ( expr )
            |   ( expr1 , ... , exprk )
            |   [ expr1 , ... , exprk ]
            |   [ expr1 [ , exp2 ] .. [ expr3 ] ]
            |   ( atomic oper )
            |   ( oper atomic )
            |   ( symoper )

expr         ::=   atomic    |   expr atomic
            |   expr oper expr        /infix operator /
            |   - expr               /negative numbers/
            |   let decls in expr
            |   expr where decls
            |   if expr then expr else expr
            |   \ variable -> expr             /lambda/

pattern      ::=   literal                    /constant/
            |   variable (a variable)
            |   ( pattern )
            |   ( pattern1 , ... , patternk ) /a tuple/
            |   [ pattern1 , ... , patternk ]
            |   constr pattern

body         ::= { impdecls ; topdecls }
```

```
            |   { impdecls }
            |   { topdecls }

topdecls    ::= topdecl1 ; ... ; topdecln
topdecl  ::= type simpletype = type
            |   data [context =>] simpletype = constrs
            |   default (type1 , ... , typen) |    decl
decls   ::= { decl1 ; ... ; decln }
decl    ::=  gendecl    |    (funlhs | pat0) rhs

gendecl  ::=  vars :: [context =>] type       /type/
            |         (empty declaration)
ops  ::=  op1 , ... , opn
vars    ::=  var1 , ... , varn
simpletype   ::=  tycon tyvar1 ... tyvark
funlhs   ::=  var apat {apat }
rhs  ->   = exp [where decls]
```

ДОДАТОК Б. ЛОГІЧНА ПРОГРАМА АЛГОРИТМУ ВИБОРУ РЕАЛІЗАЦІЙ КОНТЕЙНРІВ ДАНИХ

```
    %defines:
        implementation(hashlist).
implementation(linkedlist). implementation(treelist).
implementation(computation).
        operation(at). operation(sort).
operation(insert). operation(seqaccess).
        connection_type(ct_inherits).
connection_type(ct_convert).

        relation (recommends). relation (satisfied).
relation(unsupported).
        relation_score(satisfied, 0).
        relation_score(recommends, 1).
        relation_score(unsupported, -1).

    %integration facts:
        relation_op(seqaccess, computation,
recommends).
        relation_op(at, hashlist, recommends).
relation_op(at,treelist, satisfied).
        relation_op(sort, linkedlist, satisfied).
relation_op(sort, treelist, satisfied).
relation_op(sort, hashlist, unsupported).

    %static analysis:
```

```
    %domain rules:
        impl_fulfill(OP, IMPL, SCORE):- relation_op(OP,
IMPL, RL), relation_score(RL, SCORE), operation(OP),
implementation(IMPL), RL!=unsupported.
        impl_not_fulfill(OP, IMPL):- relation_op(OP,
IMPL, unsupported).

    %reasoning rules:
        { var_connection(VAR_FROM, VAR_TO,
CT):connection_type(CT) }1 :- var_connection(VAR_FROM,
VAR_TO).

        var_cluster_root(VAR) :- {var_connection(VAR_F,
VAR, ct_inherits): var(VAR_F)} 0, var(VAR).
        var_cluster(VAR0, VAR_TO) :-
var_connection(VAR0, VAR_TO, ct_inherits),
var_cluster_root(VAR0).
        var_cluster(VAR0, VAR_TO2) :-
var_connection(VAR_TO1, VAR_TO2, ct_inherits),
var_cluster(VAR0, VAR_TO1).
        var_cluster(VAR0, VAR0):-
var_cluster_root(VAR0).

        impl_fulfill_cluster(VAR0, OP, IMPL, SCORE) :-
var_cluster(VAR0, VAR_ANY), bind_var_op(VAR_ANY, OP),
impl_fulfill(OP, IMPL, SCORE), var_cluster_root(VAR0).
        impl_not_fulfill_cluster(VAR0,
IMPL):-var_cluster(VAR0, VAR_ANY), bind_var_op(VAR_ANY,
```

```
OP), impl_not_fulfill(OP, IMPL),
var_cluster_root(VAR0).
        bind_var_impl(VAR0, IMPL, SCORE_RESULT) :-
SCORE_RESULT = #sum[impl_fulfill_cluster(VAR0, _, IMPL,
SCORE) = SCORE], {impl_not_fulfill_cluster(VAR0,
IMPL)}0, var_cluster_root(VAR0), implementation(IMPL).

    %pretty output:
        #hide.
        #show chosen_impl/2.
        %#show bind_var_impl/3.
        %#show var_cluster_owner/1.
        %#show var_connection/3.
        %#show bind_var_op/2.
```

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ:

1. K. Czarnecki, U. W. Eisenecker Generative programming: Methods, Tools, and Applications, Addison Wesley, 2000

2. Matteo B. Generative Programming and Components: theory and practice, Addison Wesley, 2002 - 114p

3. Voelter M. A Catalog of Patterns for Program Generation, Springer, 2003 - 312p

4. Abiteboul S.m Hull R. Foundations of databases. Addison-Wesley, 1995-685p

5. Климов А., Введение в метавычисления и суперкомпиляцию, ИПС РАН, 2006 - 83с

6. Абрамов С.М, Парменова Л.В. Метавычисления и их применение . Суперкомпиляция. ИПС РАН, 2006 - 240с

7. Немытых А.П. О суперкомпиляции. ИПС РАН, 2011- 32с

7. Jones N.D. Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993. - 384p

8. Kuhne T. A Functional Pattern System for Object-Oriented Design, 1999. -328p.

9. Hughes J. Why Functional Programming matters, Addison-Wesley, 1990. -120p

10. Field A., Harrison P. G. Functional Programming, Addison-Wesley, 1988.-616p

11. Schrijvers T., Frühwirth T. Constraint Handling Rules - Current Research Topics, Springer, 2008 - 43p

12. Metakides G., Nerode A. Principles of Logic and Logic Programming. North Holland, 1996.- 344p

13. Вагин В.Н., Головина Е.Ю. и др. Достоверный и правдоподобный вывод в интеллектуальных системах. М: Физматлит, 2004. - 704с.

14. Gebser M., Kaminski R., et al. A User's Guide to gringo, clasp, clingo, and iclingo, 2010 - 82p

15. Crick, Tom. Superoptimisation: Provably Optimal Code Generation using Answer Set Programming, 2009 - 170p

16. Baral, C. Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, 2003

17. Muchnick, Steven S. Advanced Compiler design implementation, 1997