БАКАЛАВРСЬКА РОБОТА

БР. ІП - 90.00.00.000 ПЗ

Група ІПз-23-1к

Ясінський Дмитро

2025

Івано-Франківський національний технічний університет нафти і газу
Інститут інформаційних технологій
Кафедра інженерії програмного забезпечення

Ясінський Дмитро Романович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Порівняльний аналіз функціонального і об'єктно-орієнтованого програмування (назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121– Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Ясінський Д.Р.__
(підпис, ініціали та прізвище здобувача)

Науковий керівник ____ Бандура Вікторія Валеріївна, к.т.н., доцент.____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____202 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

_____

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Порівняльний аналіз функціонального і обєктно-орієнтованого програмування"

керівник проекту (роботи) Бандура Вікторія Валеріївна, к.т.н., доцент

затверджені наказом вищого навчального закладу від “ 28 ” квітня 2025 р. № 268/7

2. Строк подання студентом проекту (роботи) 10 червня 2025 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Основи та принципи об’єктно-орієнтованого програмування
2. Парадигми та проектування
3 Паралелізм та формальні моделі

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Операції визначають повідомлення (рис. 1.1, ст 10)
2. Об'єктні, класові та об'єктно-орієнтовані мови (рис. 2.4, ст 24)
3. Об'єднання об'єктів за допомогою паралелізму (рис. 3.1, ст 28)
4. (Логічно та фізично розподілені системи (рис. 3.4, ст 35)
5. Об'єкти як автомати (рис. 3.9, ст 48)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання 2025 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи		виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	25.02.2025	виконано
3	Побудова моделі або алгоритму власного рішення	15.03.2025	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	25.04.2025	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	10.06.2025	виконано

Студент _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 67 сторінок, 20 рисунків, список використаних джерел із 27 найменування,

Метою роботи Провести порівняльний аналіз функціонального та об’єктно-орієнтованого програмування з погляду теоретичних основ, парадигмального впливу на проєктування систем, гнучкості, масштабованості, підтримки абстракцій та підтримки формальних моделей.

Об'єкт дослідження: процеси проєктування та реалізації програмного забезпечення в контексті різних парадигм програмування.

Предмет дослідження: функціональне та об’єктно-орієнтоване програмування: принципи, структури, абстракції, механізми успадкування, інкапсуляції, композиції, а також формальні та паралельні моделі в обох парадигмах.

Результати дослідження: систематизовано основні характеристики та принципи функціонального та об’єктно-орієнтованого програмування.

У першому розділі - розглядаються базові поняття ООП — об’єкти, класи, успадкування та побудова об’єктно-орієнтованих систем як основи сучасної архітектури програмного забезпечення.

В другому розділі - аналізує різні парадигми програмування, включаючи функціональний підхід, типізацію, абстракції та принципи побудови ієрархій при проєктуванні програмних систем.

В третьому розділі - йдеться про теоретичні та практичні аспекти реалізації паралелізму, синхронізації та застосування формальних обчислювальних моделей у функціональному та об’єктно-орієнтованому середовищах.

Висновок: Функціональне та об’єктно-орієнтоване програмування відображають два фундаментальні підходи до побудови програмних систем.

КЛЮЧОВІ СЛОВА: ПАРАДИГМИ ПРОГРАМУВАННЯ, КЛАСИ ТА ОБ’ЄКТИ, УСПАДКУВАННЯ, ТИПІЗАЦІЯ, ПРОЄКТУВАННЯ, АБСТРАКЦІЇ

ANNOTATION

The bachelor's thesis contains 67 pages, 20 figures, a list of used sources with 27 names,

The purpose of the work To conduct a comparative analysis of functional and object-oriented programming in terms of theoretical foundations, paradigmatic influence on system design, flexibility, scalability, support for abstractions and support for formal models.

Object of research: processes of software design and implementation in the context of different programming paradigms.

Subject of research: functional and object-oriented programming: principles, structures, abstractions, mechanisms of inheritance, encapsulation, composition, as well as formal and parallel models in both paradigms.

Research results: the main characteristics and principles of functional and object-oriented programming are systematized.

The first section - considers the basic concepts of OOP - objects, classes, inheritance and the construction of object-oriented systems as the basis of modern software architecture.

The second section analyzes various programming paradigms, including the functional approach, typing, abstractions, and principles of building hierarchies in the design of software systems.

The third section discusses the theoretical and practical aspects of implementing parallelism, synchronization, and the application of formal computational models in functional and object-oriented environments.

CONCLUSION: FUNCTIONAL AND OBJECT-ORIENTED PROGRAMMING REFLECT TWO FUNDAMENTAL APPROACHES TO BUILDING SOFTWARE SYSTEMS.

ЗМІСТ

ВСТУП

У сучасному програмуванні вибір парадигми має вирішальне значення для ефективності розробки та продуктивності програмного забезпечення. Функціональне програмування та об’єктно-орієнтоване програмування (ООП) є двома ключовими підходами, які формують підґрунтя для створення надійних і масштабованих систем. Функціональне програмування акцентує увагу на чистих функціях, незмінності даних і уникненні побічних ефектів, що сприяє простоті тестування та паралельної обробки. Об’єктно-орієнтоване програмування, своєю чергою, спирається на концепції об’єктів, класів, успадкування та поліморфізму, забезпечуючи модульність і зручність моделювання складних систем.

Цей документ присвячений порівняльному аналізу функціонального та об’єктно-орієнтованого програмування з акцентом на їхні теоретичні основи, практичне застосування та вплив на проектування програмного забезпечення. У розділах, присвячених основам ООП, розглядаються ключові принципи, такі як об’єкти, класи та успадкування, які формують фундамент цієї парадигми. Парадигми проектування та типізація досліджуються через призму їхньої реалізації в обох підходах, зокрема в контексті сильно типізованих мов. Окрема увага приділяється паралелізму, де функціональне програмування демонструє переваги завдяки своїй природній підтримці асинхронних обчислень, тоді як ООП потребує додаткових механізмів синхронізації.

Актуальність теми. У сучасних умовах стрімкого розвитку інформаційних технологій та складності програмних систем вибір відповідної парадигми програмування — функціональної або об’єктно-орієнтованої — набуває особливої важливості для ефективної розробки, супроводу та масштабування програмного забезпечення.

Функціональне програмування пропонує потужний інструментарій для розробки надійних та конкурентоздатних систем з акцентом на чисті функції та паралелізм, тоді як об’єктно-орієнтований підхід забезпечує гнучку модель організації коду, що добре відображає реальні об’єкти та їх взаємодії.

Аналіз сильних і слабких сторін кожної з парадигм дозволяє глибше зрозуміти їхню роль у сучасному програмному інжинірингу, а також сформувати рекомендації щодо вибору підходу залежно від завдань проєкту.

Метою дослідження є оцінка сильних і слабких сторін кожної парадигми в різних сценаріях, включаючи продуктивність, масштабованість і зручність підтримки коду. Аналіз також охоплює формальні моделі та рефлексію, що дозволяють глибше зрозуміти теоретичні аспекти обох підходів. У висновках будуть підсумовані ключові результати порівняння та надані рекомендації щодо вибору парадигми залежно від потреб проєкту. Цей документ спрямований на створення цілісного уявлення про функціональне та об’єктно-орієнтоване програмування, що допоможе розробникам приймати обґрунтовані рішення.

На основі мети розглянемо такі завдання дослідження:

Результати дослідження: Обґрунтовано, що об’єктно-орієнтована парадигма краще підходить для моделювання складних систем з багатим набором взаємодій, тоді як функціональний підхід забезпечує високий рівень формалізму, чистоти та ефективної роботи з паралелізмом.

Об'єкт дослідження процеси проєктування та реалізації програмного забезпечення в контексті різних парадигм програмування.

Предмет дослідження: Функціональне та об’єктно-орієнтоване програмування: принципи, структури, абстракції, механізми успадкування, інкапсуляції, композиції, а також формальні та паралельні моделі в обох парадигмах.

Методи дослідження: порівняльний аналіз, метод формалізації для опису моделей паралелізму та успадкування, описово-аналітичний підхід, систематизація концепцій, метааналіз.

Бакалаврська робота містить 70 сторінок, 20 рисунків, три розділи список використаних джерел із 27 найменуванням.

РОЗДІЛ 1 ОСНОВИ ТА ПРИНЦИПИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

1.1 Об'єкти та класи

Об'єкти — це набори операцій, які мають спільний стан. Операції визначають повідомлення (виклики), на які об'єкт може відповідати, тоді як спільний стан прихований від зовнішнього світу і доступний лише для операцій об'єкта (див. рис. 1.1). Змінні s, що представляють внутрішній стан об'єкта, називаються змінними екземплярів, а їх операції - методами. Набір методів визначає його інтерфейс і поведінку.

назва: об'єкт

локальні змінні екземпляра (спільний стан)

операції або методи (інтерфейс шаблонів повідомлень, на які може відповідати об'єкт)

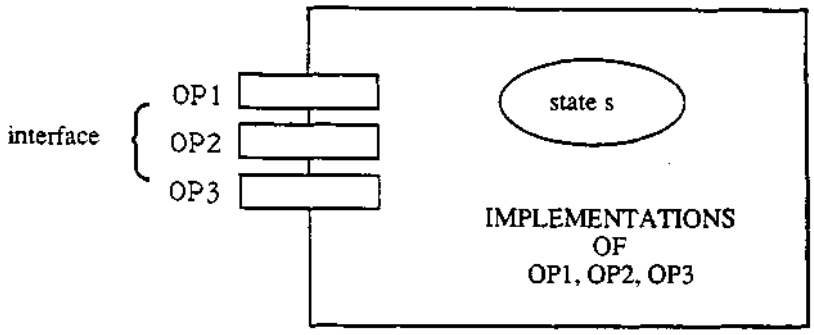


Рисунок 1.1 - Операції визначають повідомлення

Об'єкт під назвою *point* зі змінними екземплярів *x, y* і методами для їх читання та зміни можна визначити наступним чином:

```
point: object
  x := 0; y := 0;
  read-x: ↑ x;  – return value of x
  read-y: ↑ y;  – return value of y
  change-x(dx): x := x + dx;
  change-y(dy): y := y + dy;
```

Об’єкт “точка” захищає свої змінні екземпляра x,y від довільного доступу, дозволяючи доступ лише через повідомлення для читання та змінення операцій. Поведінка об'єкта повністю визначається його відповідями на прийнятні повідомлення та не залежить від представлення даних його змінних екземпляра. Крім того, знання об'єкта про його абонентів повністю визначається його повідомленнями. Об’єктно-орієнтована передача повідомлень полегшує двосторонню абстракцію: відправники мають абстрактне уявлення про

(op1, op2, ... ,opN)

одержувачів, а одержувачі мають абстрактне уявлення про відправників.

Інтерфейс операцій (методів) об’єкта може бути представлений записом:

(op1:T1, op2:T2, ... ,opN:TN)

Об’єкти, операції орі яких мають тип Ti, мають інтерфейс, який є типізованим записом:

Інтерфейси типізованих записів називаються підписами. Точковий об’єкт має таку сигнатуру:

point-interface = (*read-x*:**Real**, *read-y*:**Real**, *change-x*:**Real** → **Real**, *change-y*:**Real** → **Real**)

Операції без параметрів read-x і read -y повертають *дійсне* число як своє значення, тоді як change-x і change -y очікують *дійсне* число як аргумент і повертають *дійсний* результат.

Операції об’єкта поділяють його стан, так що зміни стану в результаті однієї операції можуть бути видимі для наступних операцій. Операції отримують доступ до стану за допомогою посилань на змінні екземпляра об’єкта. Наприклад, read-x і change-x спільно використовують змінну екземпляра x, яка є нелокальною для цих операцій, але локальною для об’єкта.

Нелокальні посилання у функціях і процедурах зазвичай вважаються шкідливими, але вони важливі для операцій всередині об’єктів, оскільки вони є єдиним механізмом, за допомогою якого операції об’єкта можуть отримати доступ до його внутрішнього стану. Спільне використання незахищених даних в

межах об’єкта поєднується з надійним захистом (інкапсуляцією) від зовнішнього доступу. Сильна інкапсуляція в інтерфейсі об’єкта реалізована за рахунок модульності (і можливості повторного використання) операцій компонентів . Це фіксує різницю всередині будь-якої організації чи організму між тісно інтегрованими внутрішніми підсистемами та визначеними договором інтерфейсами із зовнішнім світом.

Спільне використання змінних екземплярів методами можна описати за

```
let x := 0; y := 0; in
  read-x: ↑ x;
  read-y: ↑ y;
  change-x(dx): x := x + dx;
  change-y(dy): y := y + dy;
endlet
```

допомогою нотації let:

Це речення let визначає анонімний об’єкт із локальним середовищем змінних екземплярів, доступних лише для локально визначених операцій. Ми можемо розглядати об’єкти як іменовані речення let у формі « *ім’я об’єкта: речення let».* У нотації Lisp ім’я об’єкта може бути пов’язане з об’єктом за допомогою команди *визначення* у формі “(*define object-name let-clause)”* або за допомогою команди призначення у формі *“(set-q object-name let-clause)”* (див. [ASS], розділ 3).

Нотація Let спочатку була введена для функціонального програмування та вимагала, щоб локальні змінні, такі як x, y, були лише для читання. Наша нотація let принципово відрізняється від функціональної нотації let тим, що дозволяє присвоєння локальним змінним екземплярів.

Класи служать шаблонами, з яких можна створювати об'єкти. *Точка* класу має ті самі змінні екземпляра та операції, що й *точка* об’єкта , але їх інтерпретація відрізняється: тоді як змінні екземпляра об’єкта *точки* представляють *фактичні* змінні, змінні екземпляра класу є *потенційними,* екземпляри яких створюються лише тоді, коли об’єкт створюється.

точка: клас

локальні змінні екземпляра (приватна копія для кожного об’єкта класу) операції або методи (спільні для всіх об’єктів класу)

Приватні копії класу можуть бути створені за допомогою операції створення екземпляра, яка створює копію змінних екземпляра класу, на які можуть впливати операції класу.

P •'= точка створення екземпляра; — створити новий екземпляр класу point, назвати його p

Змінні екземпляра у визначеннях класу можуть бути ініціалізовані як частина створення об'єкта:

```
p1 := make-instance point (0, 0);  -- create point initialized to (0,0), call it p1
p2 := make-instance point (1, 1);  -- create point initialized to (1,1), call it p2
```

Кожна з двох точок рі, р2 має приватні копії змінних екземпляра класу та спільні операції, визначені у визначенні класу. Коли об'єкт отримує повідомлення про виконання методу, він шукає метод у своєму визначенні класу.

Ми можемо думати про клас як про визначення поведінки, спільної для всіх об'єктів класу. Змінні екземпляра вказують структуру (структуру даних) для реалізації поведінки. Публічні операції класу визначають його поведінку, тоді як приватні змінні екземпляра визначають його структуру.

1.2 Успадкування

Спадкування дозволяє нам повторно використовувати поведінку класу у визначенні нових класів. Підкласи класу успадковують операції свого батьківського класу і можуть додавати нові операції та нові змінні екземпляра.

Рисунок 1.2 описує ссавців за ієрархією успадкування класів (що представляє поведінку). Клас ссавців має підкласи людей і слонів. Клас осіб має надклас ссавців, а підкласи - студентів і жінок. Екземпляри Джон, Джоан, Білл, Мері, Дамбо мають унікальний базовий клас. Приналежність екземпляра до більш ніж одного базового класу, наприклад, Джоан, яка є студенткою та жінкою, не може бути виражена.

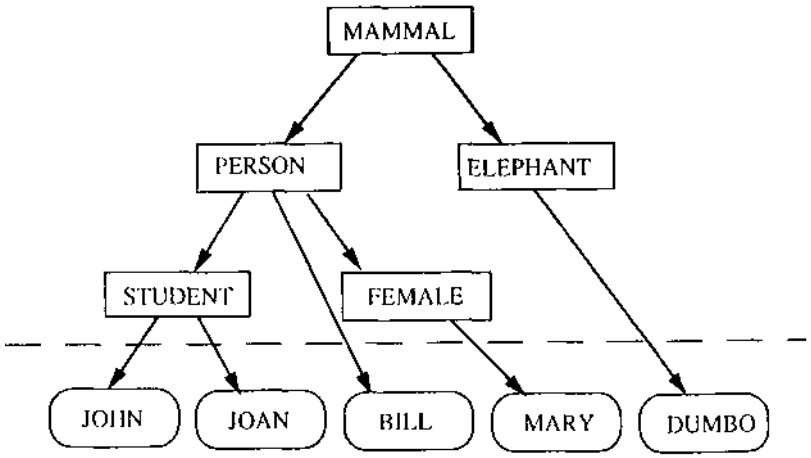


Рисунок 1.2 - Приклад ієрархії успадкування

Чому успадкування відіграє таку важливу роль в об'єктно-орієнтованому програмуванні? Частково тому, що він фіксує форму абстракції, яку ми називаємо *суперабстракцією,* яка доповнює абстракцію даних. Спадкування може виражати відносини між поведінкою, такими як класифікація, спеціалізація, узагальнення, наближення та еволюція. На рисунку 1.2 ми класифікуємо ссавців на людей і слонів. Слони спеціалізуються на властивостях ссавців, і навпаки, ссавці узагальнюють властивості слонів. Властивості ссавців наближаються до властивостей слонів. Крім того, слони еволюціонували від ранніх видів ссавців.

Спадкування класифікує класи так само, як класи класифікують значення. Здатність класифікувати класи забезпечує більшу потужність класифікації та концептуального моделювання. Класифікацію класів можна назвати класифікацією другого порядку. Спадкування забезпечує обмін, керування та маніпулювання поведінкою другого порядку, що доповнює керування об’єктами першого порядку за класами.

Альтернативний погляд на об’єкти, класи та успадкування

Ми представляємо Common Lisp Object System [CLOS], щоб надати читачеві альтернативний погляд на те, що таке об'єктно-орієнтоване програмування. CLOS має альтернативний погляд на відношення між класами,

методами та об’єктами, визначаючи класи виключно їх змінними екземплярів:

(defclass classname (список суперкласів) (список змінних екземплярів))

Методи окремо визначаються специфікаціями *defmethod* для вже визначених класів:

(метод defmethod (список класів) (визначення методу))

CLOS дозволяє класу мати кілька суперкласів (множинне успадкування), а методу мати кілька класів. Оскільки методи можуть мати кілька класів, вони не можуть бути визначені в одному класі. Замість цього всі методи, які мають задане ім’я методу, групуються разом у єдиному *визначенні defgeneric* :

(defgeneric methodname (список визначень методів із заданою назвою методу))

Екземпляри класів створюються командою *make-instance* , яка створює об’єкт зазначеного класу. Команда Lisp set-q пов’язує ім’я з новоствореним об’єктом:

(set-q inst-X (make-instance class-C)) — створити об’єкт *класу C,* назвати його *inst-X*

Коли об’єкт CLOS отримує повідомлення про виконання будь-якого методу, він звертається до загального визначення цього методу, щоб визначити його контекст і середовище виконання. Це контрастує з мовами, подібними до Smalltalk, які просто шукають метод у своєму визначенні класу.

Загальні функції є концептуально корисними для групування подібних операцій (поліморфних операцій) у пов’язаних класах, таких як оновлення для позначених, рамок і кольорових вікон. Але саме їхній механізм дії на декілька класів вимагає радикальної зміни структури програми. CLOS шукає методи за назвою методу як первинного ключа; класи, з якими асоціюються методи, є вторинним ключем. Навпаки, традиційні об’єктно-орієнтовані програми використовують клас об’єкта як первинний ключ для пошуку методів.

Ціною цієї додаткової гнучкості є втрата безпеки та інкапсуляції. Пізнє визначення методу для класу може змінити поведінку вже створених екземплярів. Методи, що посилаються на змінні екземплярів у кількох класах,

послаблюють інкапсуляцію, подібно до того, як людина, яка працює на кількох майстрів, може розповісти секрети одного іншому.

Вимога, щоб усі методи належали до одного класу та працювали під час виконання з одним об’єктом, значно посилює інкапсуляцію. Послаблення цієї вимоги збільшує гнучкість, але може створювати некеровані (схожі на спагетті) структури об’єктів. Спагетті-подібні структури можуть бути необхідними в розумних організмах, таких як мозок, але вважаються шкідливими в розробці програмного забезпечення, так само як гото вважається шкідливим. Це перший із багатьох компромісів між структурою та гнучкістю, розглянутих у цій статті. CLOS приймає точку зору, що програмістам слід надавати потужні інструменти та довіряти їм написання добре структурованих програм, тоді як інженери програмного забезпечення менше довіряють програмісту, накладаючи обмеження, які можуть призвести до втрати свободи та гнучкості.

1.3 Об’єктно-орієнтовані системи

Об’єкти пов’язані зі своїми клієнтами відношенням *клієнт/сервер* . Контракт між об’єктом і його клієнтами повинен визначати як обов’язки об’єкта, так і клієнта [WW]. Контракт може бути визначений *попередніми умовами* , які визначають обов’язки клієнтів, і *постумовами,* які визначають обов’язки об’єкта у відповіді [Me].

Об’єкти мають глобальні обов’язки щодо управління програмним забезпеченням (для підтримки гнучкої композиції об’єктів і еволюції системи), які доповнюють їхні локальні обов’язки щодо поведінки інших об’єктів. Управління об’єктами реалізується за допомогою класів, які дозволяють розглядати об’єкти як значення першого класу, і шляхом успадкування, що полегшує повторне використання специфікації інтерфейсу шляхом поступової модифікації та вдосконалення.

Об'єктно-орієнтована парадигма підтримує самоопис систем через *метаоб'єктні протоколи* та опис додатків шляхом розширення (спеціалізації)

ієрархій успадкування для опису системи [GR]. Закрито *на* самоопис. Він підтримує три типи абстракції: *абстракцію даних* для зв’язку об’єктів, *суперабстракцію* (через успадкування) для покращення поведінки та *метаабстракцію* як основу для самоопису:

абстракція даних — » інкапсуляція, суперабстракція зв’язку об’єктів — » керування об’єктами, мета-абстракція покращення поведінки — » самоопис

Універсальність об’єктів як формалізму репрезентації, моделювання та абстракції свідчить про те, що об’єктно- орієнтована парадигма є не лише корисною, але й фундаментальною.

1.4 Висновок по розділу

Успадкування є ключовим механізмом об'єктно-орієнтованого програмування, що забезпечує повторне використання коду, підтримку абстракції вищого порядку та концептуальне моделювання поведінки. Об’єктно-орієнтована парадигма, завдяки поєднанню інкапсуляції, суперабстракції та метаабстракції, створює гнучке середовище для моделювання складних систем.

РОЗДІЛ 2 . ПАРАДИГМИ ТА ПРОЕКТУВАННЯ

2.1 Парадигми поділу та переходу стану

Програма, стан і структура обчислення є взаємодоповнюючими абстракціями, які відповідно відображають точку зору розробника мови, розробника та агента виконання. Вони визначають три різні способи абстрагування від конкретних мов для визначення мовних класів із характерними способами мислення та вирішення проблем. Надійні парадигми, такі як об'єктно-орієнтована парадигма, взаємозамінно визначаються програмою, станом або структурою обчислення.

На відміну від моделі спільної пам’яті процедурно-орієнтованого програмування, об’єктно-орієнтоване програмування розбиває стан на інкапсульовані блоки, кожен з яких пов’язаний з автономною, потенційно паралельною, віртуальною машиною.

В архітектурі спільної пам’яті послідовності дій, включаючи процедури, мають спільний глобальний незахищений стан, як показано на рисунку 2.1. Процедури відповідають за те, щоб доступ до даних здійснювався лише авторизованим способом. Процеси повинні взяти на себе відповідальність за синхронізацію свого доступу до спільних даних, виконання протоколів входу та виходу до критичних регіонів, у яких знаходяться спільні дані, за допомогою примітивів синхронізації, таких як семафори [AS]. Для синхронізації потрібні правильні протоколи в кожному з процесів, які отримують доступ до спільних даних. Неправильні протоколи в одному процесі можуть порушити цілісність даних для всіх процесів.

Об’єктна парадигма розділяє стан на блоки, пов’язані з об’єктами, як показано на рисунку 2.2. Кожен блок відповідає за власний захист від доступу несанкціонованих операцій. У паралельному середовищі об’єкти захищаються від асинхронного доступу, знімаючи навантаження синхронізації з процесів, які отримують доступ до даних об’єкта.

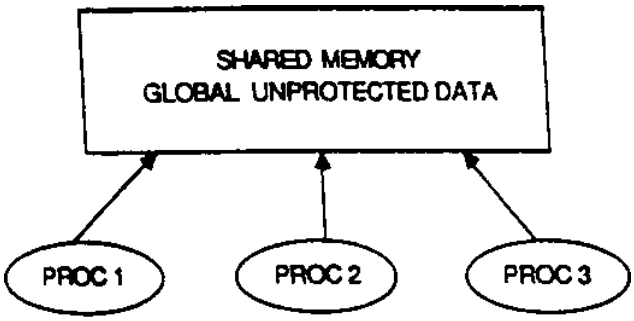


Рисунок 2.1 - Архітектури спільної пам'яті

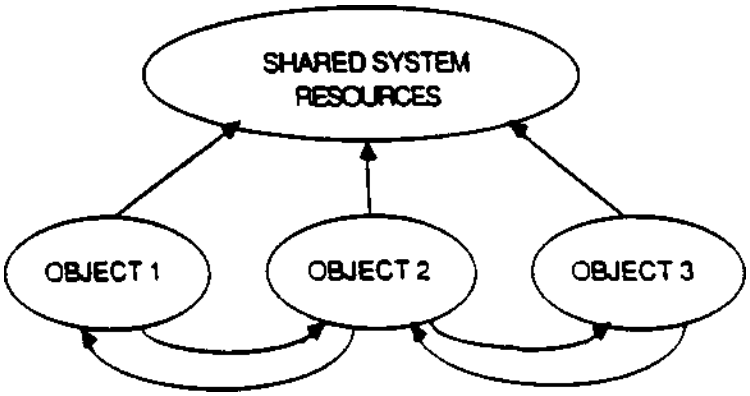


Рисунок 2.2 - Об'єктно-орієнтована, розподілена архітектура

Поділ держави на непересічні інкапсульовані частини є визначальною рисою розподіленої парадигми. Об'єктно-орієнтовані програми логічно розподілені. Але об’єктно-орієнтовані системи наголошують на інтерфейсах користувача та системному моделюванні, тоді як розподілені системи наголошують на надійному обслуговуванні за наявності збоїв, реконфігурації апаратного та програмного забезпечення та автономності володіння та контролю. Об’єктно-орієнтоване програмування наголошує на управлінні об’єктами та проектуванні додатків за допомогою таких механізмів, як класи та успадкування, тоді як розподілене програмування наголошує на паралельності, реалізації та ефективності.

Незважаючи на ці відмінності в акцентах, існує сильна спорідненість між об'єктно-орієнтованою та розподіленою архітектурами. Оскільки об’єктно-орієнтовані програми логічно розподілені, парадигму можна розширити для підтримки фізичного розподілу та одночасного виконання компонентів. В

ідеальному світі розподілена та об’єктно-орієнтована парадигми були б об’єднані, поєднуючи переваги надійності та ефективності з перевагами зручності для користувача та багаторазового використання. Але такі багатопарадигмальні системи можуть виявитися надто складними через їхню спробу обслуговувати занадто багато господарів, як PL/I та Ada.

Окремі фрагменти розділеного стану, що відповідають окремим об’єктам, мають структуру спільної пам’яті, як показано на рисунку 2.2. Об’єкт – це набір процедур, які спільно використовують незахищений стан. Ця парадигма розподілу за станом базується на парадигмі спільної пам’яті як структурі окремих компонентів програмного забезпечення та вводить другий рівень структури з зовсім іншим набором правил взаємодії для спілкування між фрагментами розділеного стану.

Парадигми переходу станів, комунікації та класифікації

Мовні парадигми можуть бути класифіковані за ступенем, до якого вони використовують перехід станів, комунікацію та класифікацію (висловлювання, модулі, типи) для обчислень (див. Рисунок 2.3).

Ми можемо думати про перехід стану, комунікацію та класифікацію як про три *виміри* простору мовного дизайну, а про мови як про точки, що займають цей тривимірний простір. Мови, розташовані поблизу осі або площини, утвореної двома осями, є низькорівневими, тоді як мови високого рівня мають збалансовану комбінацію цих обчислювальних функцій. Парадигма переходів станів розглядає обчислення як послідовність переходів станів, реалізованих виконанням послідовності інструкцій. Машини Тьюринга є перехідними

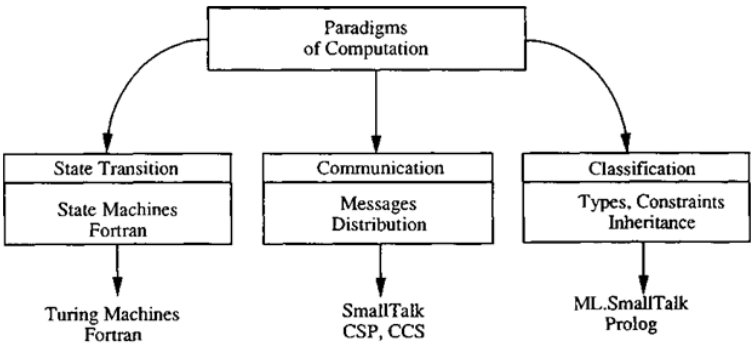


Рисунок 2.3 - Парадигми перехідного стану, комунікації та класифікації

механізми, в яких інструкції визначаються вхідним символом і поточним станом. Комп'ютери зі збереженою програмою зберігають інструкції як частину свого стану та переміщують інструкції до центрального процесора перед їх виконанням. Мови асемблера та імперативні мови вищого рівня втілюють парадигму переходу стану.

Парадигма комунікації розглядає зв'язок між агентами як основний обчислювальний механізм. Комунікаційні примітиви надсилають і отримують паралельно примітиви переходу між станами, зберігають і отримують або записують і читають. Канал зв'язку, іноді втілений буфером , відіграє роль сховища, але загалом підтримує неруйнівні операції надсилання/запису на відміну від деструктивної операції призначення парадигми переходу між станами. Ttiere — це подвійність між комунікацією та парадигмами переходу між станами, де канали відіграють роль сховища. Однак обчислення збагачуються шляхом поєднання парадигм зв'язку та переходу між станами з агентами, які мають як внутрішній стан, так і порти з чергами повідомлень.

Актори [Ag], Обчислення комунікаційних систем [Mi], Self [US] і A'UM [Yo] наближаються до чистої комунікації. Хоча обчислення на досить примітивному рівні можна розглядати як чисту комунікацію (змінні можна розглядати як канали комунікації), агенти зазвичай мають як внутрішню, так і комунікаційну поведінку. Абстракція даних підтримує комунікаційну - парадигму, наполягаючи на тому, що об'єкти визначаються виключно їх комунікаційним інтерфейсом (незалежно від їх внутрішнього стану).

Але поведінка агентів часто більш природно виражається їхньою поведінкою переходу між станами, ніж їхньою комунікаційною (інтерфейсною) поведінкою: знання внутрішньої роботи чорної скриньки часто допомагає нам зрозуміти її поведінку. Спільне використання між агентами найпростіше моделюється за допомогою моделі переходу стану (змінні зі значеннями). Актори базуються на парадигмі зв'язку, але мають імена (спільні) поштові скриньки зі станом. Потокові мови, такі як A'UM, обчислюють шляхом злиття та інших операцій із потоками, але потоки фактично мають стан (канали зв'язку – фактично змінні).

Парадигма класифікації розглядає обчислення як послідовність кроків класифікації, кожен з яких служить для обмеження результату. Повне обчислення завершується, коли послідовність кроків класифікації дає одиночні елементи. Наприклад, швидке сортування класифікує елементи вектора щодо опорного елемента та повторює цей процес для підвекторів, доки всі не стануть одноглетонними елементами. Гра «Двадцять запитань» починається з домену тварин, рослин або мінералів і ставить послідовність класифікаційних питань, призначених для зведення домену до єдиного елемента.

У той час як Quicksort і Twenty Questions виконують повне обчислення за допомогою класифікації, типи класифікують значення як прелюдію до обчислення іншими засобами. Типи класифікують значення за операціями, які до них застосовуються, щоб створити контекст для обчислення та основу для перевірки того, що під час виконання до типу застосовуються лише відповідні операції. Потім обчислення програм із перевіркою типів може здійснюватися за допомогою переходу стану та зв'язку.

Об'єктно-орієнтоване програмування підтримує класифікацію об'єктів на класи як «першого порядку», так і класифікацію класів «другого порядку» за їхніми суперкласами. Класифікація відіграє значно більшу роль в об'єктно-орієнтованих мовах, ніж у мовах, які не підтримують успадкування, оскільки класифікація класів другого порядку доповнює класифікацію об'єктів першого порядку. Ієрархії успадкування забезпечують механізм класифікації вищого

порядку, який значно збільшує виражальну силу об’єктно-орієнтованих мов.

Чисті парадигми обчислень, як правило, є низькорівневими. Наприклад, машини Тьюрінга обчислюють за парадигмою чистого переходу станів, актори за парадигмою майже чистого зв’язку, а математична теорія множин або обчислення предикатів – за парадигмою чистої класифікації. Мови високого рівня зазвичай використовують комбінацію парадигм.

Однак різні мови високого рівня по-різному поєднують функції переходу станів, комунікації та класифікації. Об’єктно-орієнтовані мови поєднують три парадигми унікальним способом, використовуючи стан переходи для обчислень всередині об’єктів, повідомлення для зв’язку між об’єктами та дворівнева класифікація для керування як об’єктами, так і класами.

Успіх об’єктно-орієнтованої парадигми пояснюється безперебійною інтеграцією переходу між станами, зв’язку та класифікації. Інструкції, модулі та типи відіграють взаємодоповнюючі та допоміжні ролі в обчислювальному процесі та підсилюють один одного, сприяючи загальному дизайну, реалізації та підтримці прикладних систем.

Парадигми можуть бути уточнені на підпарадигми, які визначають абсолютно нові способи мислення та вирішення проблем. Процедурна та об’єктно-орієнтована парадигми є підпарадигмами загальної парадигми модульного програмування, чиї способи мислення настільки різні, що спільноти розробників мови та користувачів майже не перетинаються. Підпарадигми об’єктно-орієнтованої парадигми, пов’язані з Ada, Smalltalk і акторами, також визначають дослідницькі спільноти, що не перетинаються, і рідко спілкуються одна з одною. Невеликі мовні відмінності призводять до великих змін парадигми.

Накладаючи обмеження на парадигми, ми можемо визначити підпарадигми для мовних підкласів. Нас особливо цікавлять стійкі умови, які можна легко перевірити на рівні мовної структури, а також визначають методологію програмування та стиль розв’язання задач, наприклад наступні підпарадигми та пов’язані з ними класи мови (див. рис. 2.4):

2.2 Типи та сильно типізовані мови

Об’єктно-орієнтовані мови: *клас усіх мов, які підтримують об’єкти.* **Мови на основі класів:** *підклас, який вимагає, щоб усі об’єкти належали до класу.*

Об’єктно-орієнтовані мови: *підклас, який вимагає від класів підтримки успадкування*

Об’єктні, класові та об’єктно-орієнтовані мови - це поступово менші класи мов із все більш структурованими вимогами до мови та більш дисциплінованою методологією програмування.

Об’єктно-орієнтовані мови підтримують функціональність об’єктів, але не керування ними. Мови на основі класів підтримують керування об’єктами, але не керування класами. Об’єктно-орієнтовані мови підтримують функціональність об’єктів, управління об’єктами за класами та управління класами за успадкуванням. Стиль вирішення проблем цих трьох мовних класів досить різний, щоб виправдати окрему ідентичність як різні парадигми.

Об’єктно-орієнтовані мови включають Ada, CLU, Simula та Smalltalk. Вони виключають мови Uke Fortran і Pascal, які не підтримують об’єкти як мовні примітиви.

CLU, Simula та Smalltalk також є мовами, заснованими на класах, оскільки вони вимагають, щоб їхні об’єкти належали до класів. Але Ada не базується на класах, оскільки її об’єкти (пакети) не мають типу

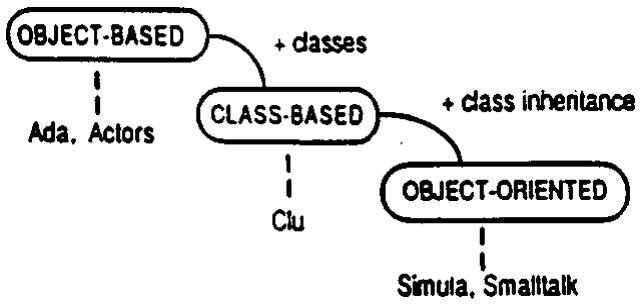


Рисунок 2.4 - Об'єктні, класові та об'єктно-орієнтовані мови

і, отже, не можуть передаватися як параметри, бути компонентами масивів чи записів або на які безпосередньо вказувати покажчики. Ці мовні переваги автоматично доступні для будь-якої типізованої сутності, але недоступні в Ada для нетипових сутностей, таких як пакети. (Зауважте, що приватні типи даних, указані в інтерфейсі пакета, мають значення першого класу, але це дещо відрізняється від розглядання самого пакета як значення першого класу. Ця невелика різниця в структурі пакета має далекосяжні наслідки для керування пакетами.)

Simula і Smalltalk є об’єктно-орієнтованими відповідно до нашого визначення, оскільки їхні класи підтримують успадкування. CLU є класовим, але не об’єктно-орієнтованим, оскільки його об’єкти мають належати до класів (кластерів), а кластери не підтримують успадкування.

Використання терміну об’єктно-орієнтований для позначення вузького класу мов, включаючи Simula та Smalltalk, але за винятком Ada та Self [Un], викликало дебати щодо правильного використання терміну об’єктно-орієнтований. Наше вузьке визначення точніше відображає об’єктно-орієнтованість у Simula та Smalltalk, ніж ширше визначення. Точність допомагає протистояти дотепі, що «всі говорять про об’єктно-орієнтоване програмування, але ніхто не знає, що це таке». Більш вільний погляд на об’єктно-орієнтоване програмування як на «будь-яку форму програмування, яка використовує інкапсуляцію» [Ni] надає довіри вищезгаданій критиці m.

Наша таксономія має практичне значення, оскільки вона розрізняє *реальні* мови, такі як Ada, Simula та Smalltalk, на основі критеріїв розробки мови, які впливають на програмування. Класифікація Ada як об'єктної, але не класової або об'єктно-орієнтованої означає, що вона підтримує функціональність об'єктів, але не керування ними, і визначає характерні відмінності в структуруванні програми та еволюції системи між Ada та Smalltalk.

Ada має багату структуру модулів, допоміжні функції, процедури, пакети, завдання та загальні модулі. Її пакунки забезпечують функціональність об’єктів, але, оскільки пакунки не мають типу, можливості Ada для керування об’єктами

є недостатніми. Ada була розроблена в той час, коли дизайн мов з абстракцією даних і паралелізмом ще не був зрозумілий, і вона не об’єднує численні поняття модульності в єдине ціле. Його поняття типу не однаково обробляє його багату і майже барокову модульну структуру. Процедури та пакети не мають типу і не можуть бути передані як параметри або відображатися як компоненти структур, тоді як завдання (конкурентні модулі) є типізованими. Той факт, що послідовні модулі не типізуються, тоді як одночасні модулі типізуються, є дещо аномальним. Відсутність підтримки керування об’єктами відображає глибші проблеми зі структурою модуля через неадекватну технологію розробки мови. Ada мала бути консервативним розширенням добре зрозумілої технології, але включення абстракції даних і паралелізму в процедурно-орієнтовану парадигму виявилося радикальним і суперечливим стрибком у незвідану територію.

Існує два типи проблем із прийняттям Ada як стандарту: її технічні обмеження та її виключення з інших законних мовних культур. Ми називаємо це проблемами *надійності* та *повноти* . Проблеми надійності вимагатимуть від користувачів більшої складності та вартості системних і прикладних програм, але їх можна пом’якшити за допомогою хороших системних інструментів. Проблеми повноти є потенційно серйознішими, оскільки вони обмежують вплив технології Ada на ширшу спільноту та відокремлюють спільноту Ada від інших мовних культур, сприяючи менталітету фортеці з психологічними бар’єрами проти багатопарадигмального програмування чи інших потенційно ефективних методів розробки програмного забезпечення.

2.3 Висновок по розділу

Об’єктно-орієнтована парадигма вирізняється інтеграцією трьох ключових підходів: переходу станів, комунікації та класифікації, що забезпечує ефективне моделювання складних систем. Завдяки інкапсуляції стану, обміну повідомленнями та ієрархічній класифікації об’єктів і класів вона дозволяє створювати надійні, гнучкі та багаторазові програмні рішення.

РОЗДІЛ 3. ПАРАЛЕЛІЗМ ТА ФОРМАЛЬНІ МОДЕЛІ

3.1 Моделі паралелізму та синхронізації

Об’єктно-орієнтоване паралельне програмування поєднує в собі об’єктно-орієнтовану та паралельну парадигми. Він поєднує в собі об’єктні поняття інкапсуляції, класів і успадкування з одночасними концепціями потоків, синхронізації та зв’язку (див. рис. 3.1). Він поєднує потужність моделювання через об’єктну орієнтацію з обчислювальною потужністю через паралелізм [Yol],

Як слід змінити послідовну об’єктно-орієнтовану парадигму, щоб пристосуватись до concurrency? Чи є паралельні об’єкти (процеси) все ще розпізнаваними об’єктами? Які типи інтерфейсів повинні мати процеси та якою повинна бути їх внутрішня структура? Чи повинні паралельні об’єкти мати класи та успадкування? Яким чином методологія проектування та методи керування об’єктами для одночасних об’єктів пов’язані з такими для послідовних об’єктів?

Об’єкти добре взаємодіють із паралельним виконанням, оскільки їхня логічна автономія робить їх природною одиницею для одночасного виконання. Однак одночасний спільний доступ є більш складним, ніж послідовний, вимагає взаємного виключення та тимчасової атомарності. Інтерфейси, внутрішня структура та протоколи зв’язку паралельних об’єктів є більш складними. Інтерфейси мають черги повідомлень, внутрішня структура може бути паралельною, а протоколи повідомлень можуть бути синхронними (клієнт/сервер), квазіконкурентними (сопрограма) або асинхронними (з ф’ючерсами).

Метою як послідовного, так і одночасного об’єктно-орієнтованого програмування є пряме та природне моделювання реального світу. Реальний світ є паралельним, а не послідовним. Об’єктний паралелізм дозволяє природним чином моделювати реальний паралелізм програм, що значно розширює наші можливості моделювання. Паралельність додає додатковий вимір складності до

програм, тому наслідки поганої структури програми, ймовірно, будуть набагато серйознішими, ніж для послідовних програм. Управління паралельними програмами через інкапсуляцію та методологію програмного забезпечення є навіть більш критичним, ніж для послідовних програм.

Ми розглядаємо питання проектування для об’єктно-орієнтованого паралелізму в таких областях:

Структура процесу Паралельність внутрішнього процесу Синхронізація Передача повідомлень Міжпроцесовий зв’язок Деталізація модулів Постійність і транзакції

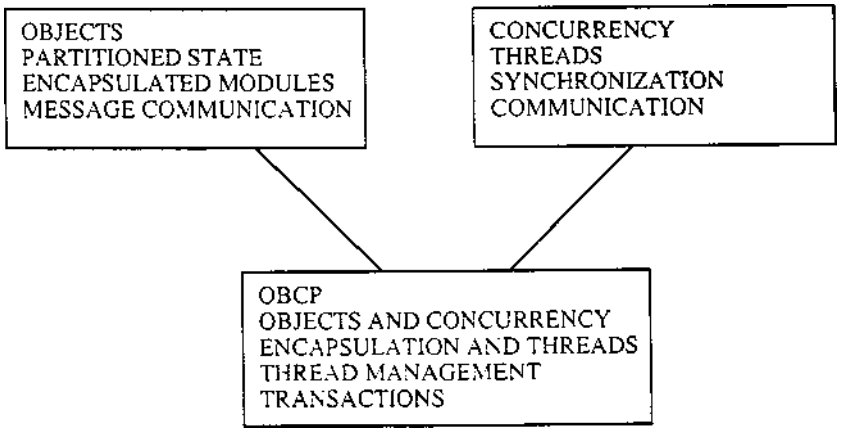


Рисунок 3.1 - Об’єктно-орієнтоване паралельне програмування

Процеси - це активні об'єкти, які повинні синхронізуватися з повідомленнями, що надходять від інших активних об'єктів. Проблеми дизайну для процесів включають: захист та інкапсуляцію, завдання проти моніторів, логічне проти фізичного розподілу та слабке проти сильного розподілу.

Розглянемо операційну систему з процесами READ, EXECUTE і PRINT, де процеси READ і EXECUTE спільно використовують вхідний буфер, а процеси EXECUTE і PRINT спільно використовують вихідний буфер, як показано на рисунку 3.2.

У моделі спільних даних дані у вхідних і вихідних буферах незахищені. Процеси введення та виконання повинні спільно захищати дані, наприклад, семафорами з операціями запиту та звільнення (P та V), які забезпечують

взаємовиключний доступ.

В об’єктно-орієнтованій моделі буфери введення та виведення є серверними процесами, відповідальними за власний захист. Процесам введення та виконання (клієнтам) більше не потрібно використовувати примітиви низького рівня для захисту даних у вхідному буфері. Можна використовувати віддалені виклики процедур, які покладаються на локальний захист даних у викликаних програмах. Завдання та монітори Ada ілюструють дві різні форми взаємодії між викликаючими клієнтами та викликаними серверними процесами.

Завдання Ada мають єдиний потік керування, який може синхронізуватися з вхідними викликами процедур у точках входу, визначених операторами accept:

Тіло завдання T приховані локальні змінні

послідовність виконуваних операторів, які включають

оператори приймання (точки входу), синхронізація шляхом зустрічі з викликами віддаленої процедури, тіла приймання, які визначають зв’язок під час кінцевого завдання синхронізації

Синхронізація вимагає, щоб викликаючий і викликаний потоки зустрічалися вчасно (див. Рисунок 3.3). Віддалені виклики процедур модуля, що викликає, повинні зустрічатися з операторами accept викликаного модуля. Якщо виклик надходить до того, як оператор прийняття готовий його прийняти, процедура виклику призупиняється, а виклик ставиться в чергу. Якщо оператор accept досягнуто перед викликом, що очікує, завдання призупиняється та очікує на виклик. Коли відбувається рандеву, нитки в

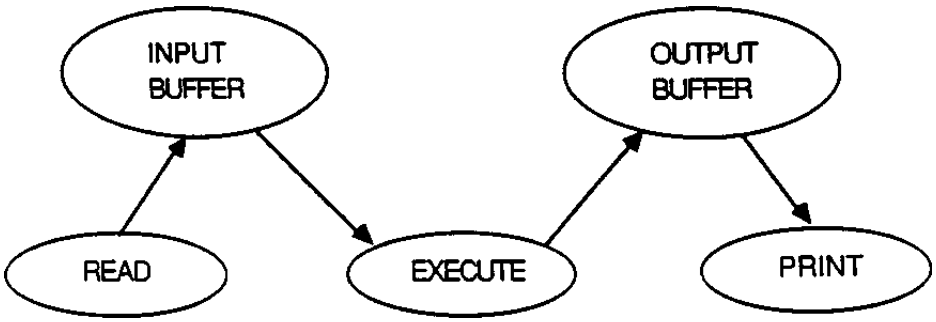


Рисунок 3.2 - Проста операційна система з вхідними та вихідними буферами

виклик і викликаний модуль тимчасово об'єднані, вхідні параметри можуть бути передані від виклику до викликаної процедури, код, визначений оператором accept, виконується, а вихідні параметри можуть бути передані назад до викликаючого. Після завершення зустрічі викликаючий і викликаний процеси відновлюють окреме одночасне виконання.

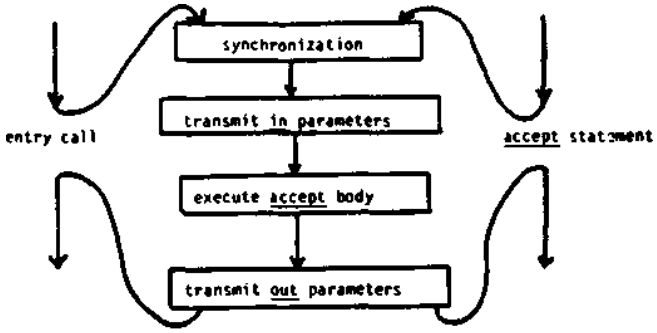


Рисунок 3.3 - Синхронізація потоків

Точки входу для операторів прийняття визначено в *специфікаціях завдання.* Буферне завдання з точками входу APPEND і REMOVE можна вказати таким чином:

завдання БУФЕР є

запис ДОДАТИ (М: вхід ПОВІДОМЛЕННЯ) запис ВИДАЛИТИ (М: вихід ПОВІДОМЛЕННЯ) кінець БУФЕР

У тілі завдання є оператори accept для прийняття викликів APPEND і REMOVE:

прийняти APPEND (М: у ПОВІДОМЛЕННІ) do BUFF(IN) := М; кінець

Код між do і end виконується під час зустрічі. У цьому випадку жодне значення не повертається абоненту (М є параметром *in*). Однак оператор accept для REMOVE має *вихідний* параметр, який повертає значення М його викликаючому як останню дію рандеву:

прийняти ВИДАЛИТИ (М: вийти ПОВІДОМЛЕННЯ) do М := BUFF(OUT); кінець

Ми можемо розрізняти *серверні завдання* з операторами *accept* , але без викликів інших завдань, і *клієнтські завдання* з викликами інших завдань, але без операторів *accept* . Завдання буфера є типовим сервером, тоді як завдання

виробника, які надсилають свої результати в буфер, є прикладами клієнтів. Завдання, які одночасно надають послугу та викликають інші завдання, діють і як сервери, і як клієнти.

Монітори мають локальні змінні та інтерфейс операцій, які фактично є точками входу. Вони гарантують взаємовиключний доступ клієнтів до спільних даних, але їхні операції можуть призупинятися та пізніше відновлюватися (як співпрограми):

Монітор M

приховані локальні змінні

операції (точки входу), які включають

команди очікування для призупинення та команди сигналу для відновлення операцій endmonitor

Монітори мають чергу входу для вхідних викликів монітора та черги очікування для призупинених викликів монітора. Призупинення реалізується за допомогою команди очікування (назва-умови), яка призупиняє поточний потік і розміщує його в іменованій черзі очікування, з якої її можна видалити (повторно пробудити) командним сигналом (назва-умови). Коли потік призупиняється, потік, що очікує, пробуджений або входить, може розпочати виконання.

Буфер з операціями APPEND і REMOVE і внутрішніми чергами очікування EMPTY і FULL проілюстровано на рисунку 3.3. Спроба виконати REMOVE, коли буфер порожній, призведе до того, що вхідний потік буде розміщено в порожній черзі очікування, а спроба виконати APPEND, коли буфер заповнений, призведе до розміщення потоку в черзі очікування FULL.

Монітори контролюють виконання потоків більш гнучко, ніж завдання, дозволяючи потокам у модулі призупиняти та пізніше відновлювати. Це вимагає більшої складності на мовному рівні з командами очікування та сигналу, а також більшої складності реалізації з внутрішніми чергами моніторингу. Система вказує пріоритети для відновлення сигналізованих потоків у чергах очікування та вхідних потоків у черзі моніторингу: процеси в чергах очікування зазвичай мають пріоритет над вхідними процесами в черзі моніторингу.

Недетермінізм є необхідною властивістю будь-якої клієнт-серверної системи, оскільки вимоги клієнтів до послуг неможливо передбачити. Об’єкти та монітори залежать від клієнтів, щоб вибрати наступну операцію для виконання. Об’єкт BUFFER не може знати, чи буде наступна операція APPEND або REMOVE.

Завдання Ada пристосовуються до недетермінованої поведінки клієнтів за допомогою оператора select, який вибирає серед захищених викликів віддалених процедур, що очікують (rpcs):

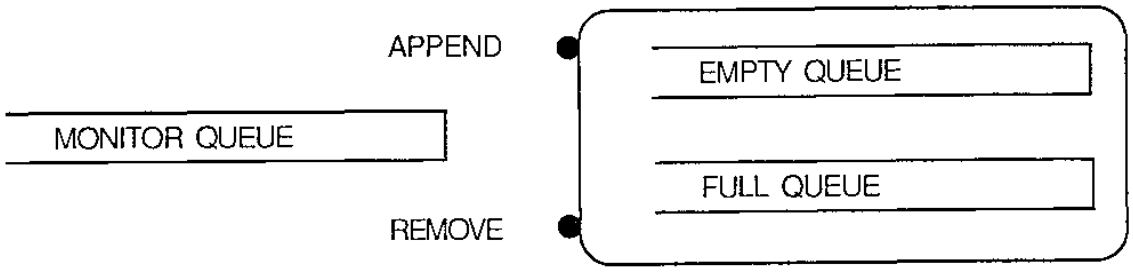
вибрати

коли guard-conditionl приймає rpcl

або

коли guard-conditionl приймає rpcl

коли guard-conditionN тоді прийняти rpcN endselect



Оператор Select можна використовувати для реалізації недетермінованого вибору між операціями APPEND і REMOVE буферного завдання Ada:

вибрати

якщо неповний, тоді прийміть APPEND або

коли не пусто, тоді прийміть ВИДАЛИТИ endselect

Коли буфер не заповнений і не порожній, цей оператор вибирає одну з операцій APPEND і REMOVE, що очікують, а в інших випадках вибирає першу, яка надійшла. Коли буфер заповнено, виконання REMOVE розблокує буфер для APPEND. Для порожнього буфера виконання APPEND розблокує його для REMOVE.

Явний оператор select Ади контрастує з неявним недетермінізмом моніторів, які складаються з неявного оператора select із незахищеними альтернативами:

виберіть opl або op2 або ... або opN і завершіть вибір

Захищений вибір реалізується в моніторах операторами *очікування* (wait(guard-condition)), які можуть призупинити операції монітора під час їх виконання. Монітори є більш гнучкими, ніж завдання Ada, оскільки дозволяють охоронцям виникати не лише при вході в процес, але й у будь-який момент під час виконання процесу. Вони відокремлюють недетермінований запис від захисних умов, які визначають, чи готовий процес до виконання.

Недетермінізм вбудований у тканину об'єктно-орієнтованого програмування, незалежного від concurrency. Недетермінізм об'єктів відображає реальний світ, де послідовність подій, у яких беруть участь суб'єкти, неможливо передбачити. Оскільки недетермінізм виникає навіть у послідовному випадку, недетермінізм об'єктно-орієнтованого паралелізму приходить безкоштовно.

Недетермінізм об'єктів відрізняється від традиційних недетермінованих автоматів, де недетермінізм відноситься до того факту, що крок обчислення в даному стані може генерувати кілька наступних станів. Об'єкти мають недетермінований вхід для кроку обчислення, тоді як недетерміновані автомати мають недетермінований вихід. Дві форми недетермінізму є подвійними у значенні, і їх можна назвати недетермінізмом входу та виходу.

недетермінований вхід: *наступна операція над об'єктом або процесом невідома* недетермінований вихід: *вихід є недетермінованою функцією вхідних даних*

Інструкції Select виникають через необхідність обробки недетермінованості вхідних даних клієнтів. Вони підтримують недетермінований вихід, коли кілька гілок оператора select готові до виконання. Наприклад, якщо інструкція select завдання буфера досягнута, коли буфер не порожній і не заповнений, а виклики APPEND і REMOVE очікують на виконання, тоді для визначення того, яка дія фактично виконується, можна

використовувати вихідні дані no ndeterminism.

Система модулів є логічно розподіленою, якщо кожен модуль має свій окремий простір імен. Інші модулі не мають безпосереднього доступу до локальних даних, і навпаки, модулі не можуть отримати прямий доступ до нелокальних даних і повинні спілкуватися з іншими модулями за допомогою повідомлень.

Ми розрізняємо логічний розподіл, визначений у термінах властивостей простору імен, і фізичний розподіл, визначений у термінах географічного чи просторового розподілу модулів. Фізичний розподіл зазвичай передбачає логічний розподіл, оскільки фізичне розділення є найбільшим природним чином моделюється логічним розділенням. Але логічно розподілені системи часто реалізуються за допомогою архітектур спільної пам’яті для більшої ефективності. Об'єктно-орієнтовані системи є логічно розподіленими, але зазвичай реалізуються на нерозподілених комп'ютерах. Зв’язок між логічно та фізично розподіленими процесами проілюстровано на рисунку 3.4.

Логічний розподіл підтримує автономність програмних компонентів і тим самим полегшує поточне виконання. Іншою важливою перевагою логічного розподілу є підтримка модулів автономного інтерфейсу, таких як кілька вікон. Модулі автономного інтерфейсу дозволяють користувачеві виконувати декілька автономних, концептуально одночасних дій інтерфейсу. Фізична одночасність не потрібна і зазвичай непотрібна. Але концептуальна автономія є важливою властивістю діяльності в реальному світі, і її реалізація в інтерфейсі робочої станції є однією з найважливіших практичних переваг об’єктно-орієнтованого програмування.

Система є слабко розподіленою, якщо її модулі знають імена інших модулів. Він сильно розподілений, якщо його модулі не можуть безпосередньо назвати інші модулі. У сильно розподіленій системі даний модуль знає лише імена своїх власних комунікаційних портів. Назви модулів, з якими він спілкується, зберігаються як дані в його комунікаційних портах і можуть розглядатися як покажчики на порти інших модулів, які не можна розіменувати

для визначення імені нелокальних портів. Модуль спілкується, надсилаючи повідомлення, визначене як значення даних, до локального порту для передачі до пункту призначення.

Традиційні об'єктно-орієнтовані мови поширені слабо. Об’єкти можуть знати імена інших об’єктів і надсилати повідомлення методам інших об’єктів. Слабкий розподіл відображає те, що реалізація зазвичай відбувається в архітектурі спільної пам’яті. Сильно розподілені системи краще охоплюють реальність фізичного розподілу, де зв’язки між модулями мережі реалізуються фізичними каналами, а топологія мережі може змінюватися.

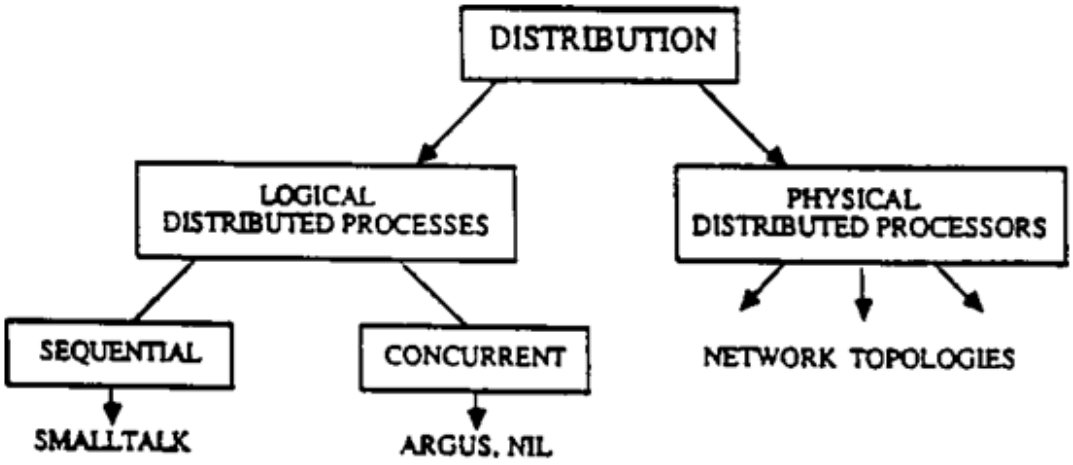


Рисунок 3.4 - Логічно та фізично розподілені системи

У сильно розподілених системах, таких як Nil і Hermes [SY], процеси можуть посилатися на власні порти, але не на порти інших процесів. Коли створюється процес Q, процес створення повинен надати Q початкові можливості, які ми тут називаємо пуповиною (див. Рисунок 3.5). Створений процес зберігає початкові можливості за допомогою команди отримання, як у програмі PR нижче, а потім може використовувати можливості для доступу до свого середовища.

Щойно створений процес знаходить свої початкові можливості в системній змінній *init*. Наведена нижче програма PR для друку повідомлення «Hello World» спочатку виконує команду «receive Port from Init», яка зберігає свої

початкові можливості в локальному комунікаційному порту під назвою *Port.* Потім він викликає компонент Putline свого комунікаційного порту з повідомленням для друку «Hello World». Нарешті, він виконує зворотне повідомлення, яке повертає його можливості абоненту та дозволяє закінчити процес.

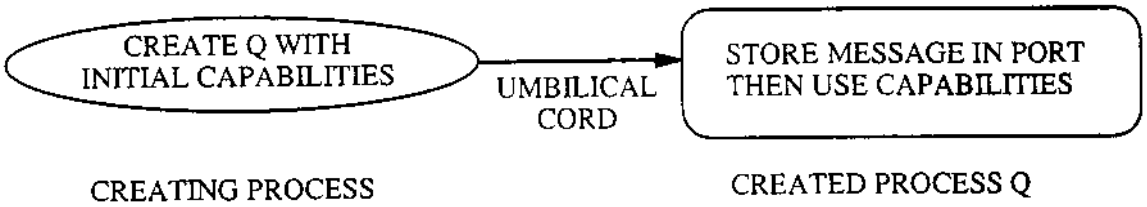


Рисунок 3.5 - Створення сильно розподілених процесів

PR: отримати порт від Init

виклик порту Putline ("Hello", "World") return Port;

Сильно розподілені системи мають більші накладні витрати, ніж слабко розподілені системи, але дозволяють динамічну реконфігурацію для анонімних (*аутичних*) процесів.

Процеси об’єктно-орієнтованих паралельних систем можуть бути внутрішньо послідовними, квазіконкурентними або повністю паралельними, як показано на рисунку 3.6.

Послідовні процеси проілюстровано Ada, чиї завдання мають один виконуваний потік, який може бути призупинено під час очікування отримання зовнішнього зв’язку, але повинен виконуватися до

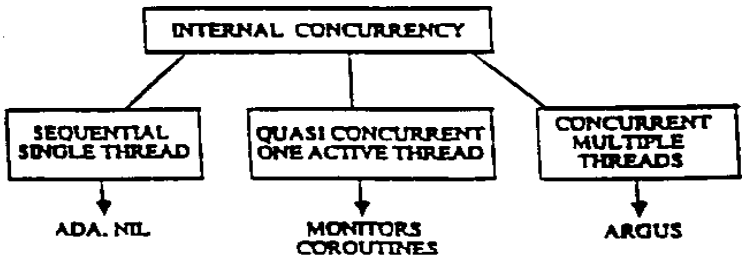


Рисунок 3.6 - Внутрішня паралельність у межах процесів

завершення після прийняття зовнішнього зв’язку. Rendezvous викликає

тимчасове злиття вхідного та виконуваного потоків, доки взаємодія двох потоків не завершиться.

Квазіконкурентне виконання ілюструється моніторами, які мають щонайбільше один активний виконуваний потік, але можуть мати призупинені потоки в одній або кількох чергах очікування. Можливість внутрішнього призупинення потоків забезпечує гнучкість, а той факт, що щонайбільше один потік може бути активним, забезпечує взаємовиключний доступ до локальних даних. Через ці переваги об'єктно-орієнтовані конкурентні мови, такі як ABCL1 і Orient 84 [YT], засновані на квазіпаралельності, хоча їхні протоколи передачі повідомлень виходять за межі механізму клієнт/сервер моніторів. Однак квазіпаралельність створює проблеми, коли потоки, що виконуються, представляють транзакції, до даних яких можна отримати доступ лише в стабільному стані. Призупинення потоків у нестабільному стані дозволяє втручатися в дані, що порушує умови транзакцій.

Повністю паралельні процеси проілюстровано Argus guardians [LS]. Опікуни не синхронізують вхідні потоки під час входу, тому введення потоку в Guardian просто збільшує кількість потоків, що виконуються. Синхронізація відбувається пізніше під час виконання потоків у межах спроби Guardian отримати доступ до спільних даних.

На рисунку 3.7 показано поштову систему Argus із опікунами MAILER, MAILDROP і REGISTRY, які можуть бути відтворені в багатьох фізичних місцях. Mailer - це інтерфейс користувача, який дозволяє користувачам надсилати пошту, отримувати пошту та додавати користувачів до поштової системи. Діан MAILDROP містить підмножину поштових скриньок і викликається MAILER для доставки пошти до цих поштових скриньок. Опікун РЕЄСТРУ вказує адресу пошти для кожного користувача, і її потрібно оновлювати, коли користувачів додають або видаляють із системи.

У цьому прикладі опікуни MAILER можуть вільно виконувати кілька потоків, оскільки немає спільної структури даних, про яку варто турбуватися. Опікуни MAILDROP повинні синхронізуватися для доставки та видалення з

певної поштової скриньки, але можуть одночасно взаємодіяти з різними поштовими скриньками. Опікуни REGIS TRY можуть дозволити одночасний пошук, але мають виконувати синхронізацію під час додавання або видалення користувачів. Затримуючи синхронізацію, щоб вона відбувалася в точці доступу до спільних даних, а не під час входу в паралельний модуль, ми можемо реалізувати більший і більш детальний паралелізм.

Ми розрізняємо синхронізацію для незахищених і інкапсульованих даних. Для незахищених даних роботу синхронізації має виконувати кожен процес, який отримує доступ до даних. Для синхронізації можуть знадобитися кооперативні протоколи: наприклад, між семафорами для визначення

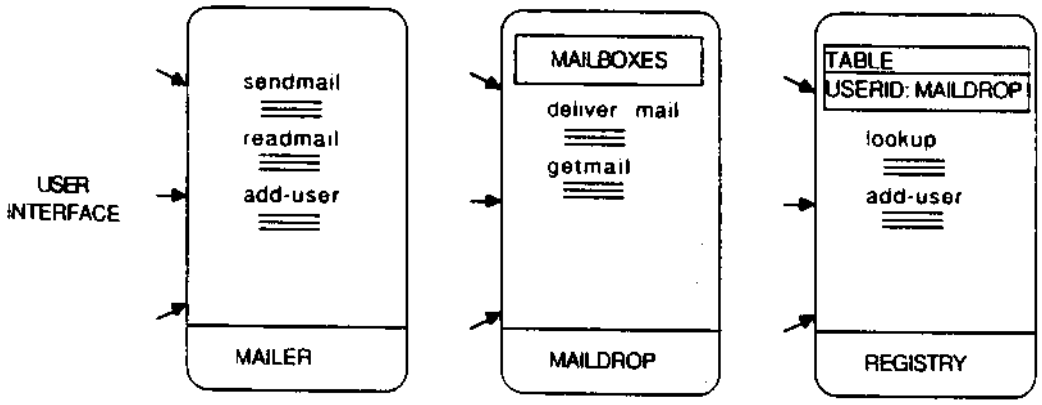


Рисунок 3.7 - Поштова система Argus

взаємовиключний доступ до критичних регіонів. Захищені дані беруть на себе відповідальність за власний захист, знімаючи навантаження з процесів, які отримують доступ до даних. Об'єктно-орієнтовані системи зосереджені на синхронізації захищених даних.

Можна виділити три типи механізмів синхронізації:

рандеву; *синхронізація між двома потоками (для послідовних процесів)*

змінні умови; *керується операціями очікування та сигналу (для квазіконкурентних процесів)*

блокування; *синхронізація між потоком і спільними даними (для повністю паралельних процесів)*

Механізм зустрічі для послідовних процесів можна розглядати як

синхронізацію між двома потоками, а саме викликаючим і викликаним потоком. Синхронізація є симетричною для двох сторін, які синхронізуються. Але викликаючий і викликаний потоки відіграють різні ролі після того, як відбулася синхронізація, причому викликаючий потік є пасивним, а викликаний потік виконує завдання, для якого він був викликаний. Зустріч Ada спричиняє тимчасове об'єднання викликаного та викликаного потоків з метою виконання тіла прийняття та подальше розгалуження, щоб дозволити відновити одночасне виконання викликаного та викликаного процесів.

Квазіконкурентні процеси використовують неявну синхронізацію, подібну до рандеву, щоб визначити вхід до монітора, і використовують змінні умови (сторожі) для моделювання внутрішньої синхронізації. Вони роз'єднують синхронізацію для входу та відновлення потоків, маючи різні протоколи для входу та відновлення, визначені різними мовними примітивами.

Повністю паралельні процеси передбачають синхронізацію між паралельно виконуваними потоками та локальними незахищеними спільними даними. Це реалізується шляхом блокування даних і тимчасового обмеження - доступу до потоку, від імені якого дані були заблоковані. Протокол блокування сам по собі може бути складним, коли операція вимагає ексклюзивного доступу до кількох спільних об'єктів даних. Двофазне блокування, яке відокремлює фазу отримання блокувань від фази їх звільнення, зменшує ймовірність поразки в конкуренції потоків за кілька спільних ресурсів. Альтернативні протоколи блокування для розподілених архітектур обговорюються в [GT].

Альтернативи дизайну для передачі повідомлень включають *синхронну, асинхронну* та *потокову* передачу повідомлень. Синхронна передача повідомлень, яка вимагає від відправника призупинення до отримання відповіді, по суті, є віддаленим викликом процедури. Асинхронна передача повідомлень дозволяє відправнику продовжити, але вимагає синхронізації, якщо подальше виконання залежить від відповіді. Передача повідомлень на основі потоку підтримує потоки повідомлень, які так само потребують синхронізації під час використання повідомлення, щоб перевірити, чи отримано відповіді.

Асинхронно обчислені результати можуть оброблятися структурами даних, які називаються *ф'ючерсами* , створеними під час надсилання повідомлення. Анонімні ф'ючерси дозволяють синхронізувати оператори для асинхронно обчислених аргументів і можуть бути призначені змінним:

+(future(el), future(e2)) — зачекайте, поки el і e2 будуть готові, перш ніж додавати define(x, future(e)) — x не можна використовувати, доки не буде завершено оцінювання e

+ вимагає *суворого* (синхронного) обчислення, яке блокується, доки не будуть обчислені його аргументи та сума, *define* дозволяє *нестроге* (ледаче, асинхронне) обчислення, яке блокується лише тоді, коли x здійснюється доступ для подальшого обчислення.

Асинхронне *надсилання* можна розглядати як несувору операцію, яка створює *змінну відповіді* під час надсилання повідомлення для синхронізації під час використання повідомлення:

send(message) x(future(reply)) — створити x під час надсилання для синхронізації під час використання

Майбутні об'єкти використовуються в об'єктно-орієнтованих системах, таких як ABCL [Yol], вони використовуються в механізмі *потокового виклику системи* МГГ Mercury [LBGSW], який дозволяє відправнику надсилати потік повідомлень уздовж потоку викликів, а одержувачу надсилати зворотний потік відповідей (див. Рисунок 3.6). Синхронізація для потокових викликів реалізована в Argus за допомогою типізованих структур даних, які називаються *обіцянками* [LSI], які, як і ф'ючерси, створюються під час виклику (надсилання повідомлення) і можуть бути витребувані лише тоді, коли обіцянка виконана (відповідь збережена в обіцянці). Mercury підтримує синхронну, асинхронну та потокову передачу повідомлень для гетерогенних, взаємодіючих розподілених процесів:

Mercury Communication Mechanisms : *синхронний віддалений виклик процедури* ; *асинхронне надсилання з поверненням для винятків* streamcall: *конвеєрний потік викликів і повернень (Рисунок 3.6)*

Комунікація між процесами

Альтернативи дизайну для зв’язку між процесами включають двосторонні взаємопов’язані розподілені процеси, односторонні взаємопов’язані клієнт/серверні процеси та динамічно взаємопов’язані сильно розподілені процеси.

У зв’язку CSP потрібне двостороннє присвоєння імен, коли процес-відправник знає ім’я процесу-одержувача, а процес-одержувач – ім’я процесу-відправника, як показано на рисунку 3.6. Двостороннє іменування моделює жорстке з’єднання між процесами. Однак, одностороннє іменування, коли викликаному процесу не потрібно знати свої імена абонентів, є більш гнучким і є стандартним міжмодульним механізмом зв’язку для процедур і синхронної передачі повідомлень. Двостороннє іменування реалізовано в термінах одностороннього іменування шляхом передачі імені абонента як аргументу викликаної процедури. Інструкції Ada *accept* використовують одностороннє іменування, вимагаючи від абонента назвати викликану задачу, але дозволяючи викликати задачу будь-кому, хто знає її назву (див. Рисунок 3.7).

У сильно розподілених процесах імена нелокальних портів зберігаються як дані в локальних змінних порту, тому підключення до інших процесів є динамічним. Взаємозв’язки каналів встановлюються шляхом збереження значень портів у змінних порту, як показано на рисунку 3.8. Канал можна розглядати як кабель із штекером на одному кінці, що з’єднує його з метою, і гніздом на іншому кінці, до якого можна підключити джерело.

Межі модулів в об’єктно-орієнтованих паралельних системах визначаються механізмами для абстракції, розподілу та синхронізації, як показано на рисунку.

Межа абстракції - це інтерфейс, який модуль представляє своїм клієнтам. Він визначає форму, в якій ресурси, надані об'єктом, можуть бути доступні (викликані). Це межа приховування інформації, з якою стикається клієнт, дивлячись всередину модуля. Він обмежує те, що може бачити клієнт, приховуючи локальні дані від доступу клієнта. Це одиниця інкапсуляції та

фундаментальна одиниця гранулярності об’єкта.

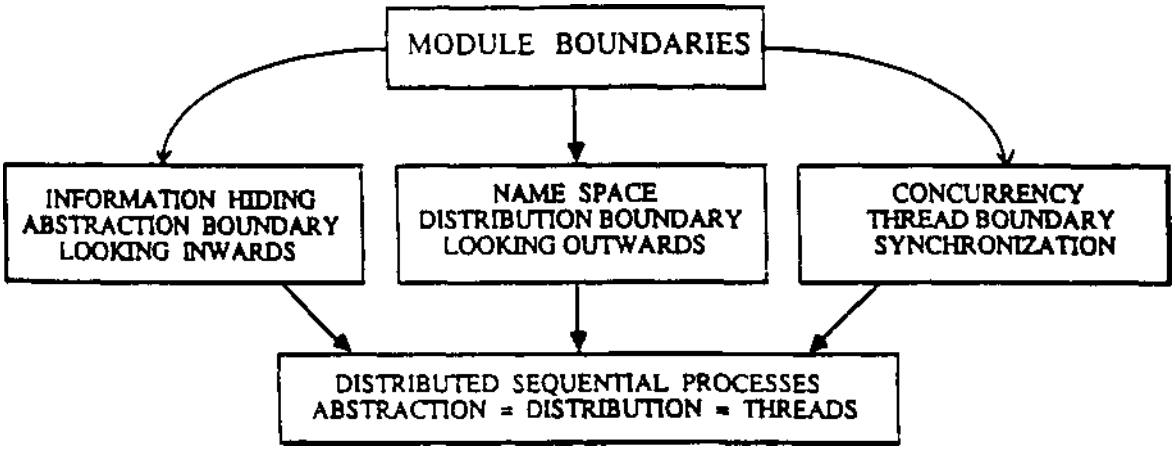


Рисунок 3.8 - Межі абстракції, розподілу та синхронізації

Межа *розповсюдження* — це межа доступних імен, видимих з об’єкта. Межа абстракції зустрічається користувачем, який дивиться всередину під час розподілу агент зустрічає межу всередині модуля, дивлячись назовні. Межа розподілу може бути більш грубою, ніж границя абстракції, як у мовах блокової структури, або лінійної, коли велика абстракція (скажімо, система бронювання авіакомпаній) реалізована розподіленими компонентами. Коли межі абстракції та розподілу збігаються, ми говоримо, що модуль *розподілений.* Абстракція для розподілених модулів є симетричною: погляд одержувача на відправника настільки ж абстрактний, як і погляд відправника на одержувача.

Межа синхронізації модуля — це межа, на якій потоки, що входять до модуля, синхронізуються з поточними діями в модулі. Для послідовних процесів межа синхронізації потоку також є межею абстракції. Для паралельних процесів межа синхронізації потоку є тоншою, ніж межа абстракції. І навпаки, одиниця абстракції може бути грубішою, ніж для паралелізму, наприклад, коли адресний простір, пов’язаний з одним потоком, може містити багато абстрактних об’єктів.

Межі для абстракції, розподілу та синхронізації загалом є незалежними. Однак окремий випадок, коли три межі збігаються, є цікавим. Процеси, для яких ці межі збігаються, називаються *розподіленими послідовними процесами.* Вони розподілені, тому що їх межі абстракції та розподілу збігаються, і послідовні, оскільки їх межі абстракції та синхронізації збігаються.

Розподілені послідовні процеси привабливо прості. Однак наполягання на однаковій деталізації для абстракції, розподілу та синхронізації може знизити ефективність або виразність. Наприклад, великі абстракції, такі як системи бронювання авіакомпаній, потребують точної синхронізації на рівні окремих рейсів або навіть окремих місць для ефективної роботи. Навпаки, мережа послідовних комп'ютерів із декількома об'єктами на кожному вузлі природним чином моделюється одиницею паралельності, грубішою за її одиницю абстракції.

Дані є постійними, якщо їх термін служби тривалий відносно операцій, які на них впливають: наприклад, дані про податок на прибуток, які зберігаються з року в рік. Однак постійність є відносною, а не абсолютною: якби податок на прибуток обчислювався щогодини (для біржових брокерів, що швидко рухаються), то постійність даних вимірювалася б годинами, а не роками.

Рисунок 40 класифікує модулі з точки зору відносної стійкості операцій і даних:

функції; *постійні дії над тимчасовими* **об'єктами даних;** *операції та дані з одночасними* **транзакціями тривалості життя,** *гнучка відносна постійність операцій і даних*

Функції та процедури підкреслюють стійкість програм для тимчасових даних. Об'єкти частково виправляють баланс, підтримуючи коектенсивну стійкість для операцій і їхніх даних, але не є гнучкими в підтримці змінної стійкості операцій і даних. Бази даних вирішують проблему збереження даних, повністю відокремлюючи програми від даних, з якими вони працюють, і запроваджують транзакції для гнучкого тимчасового зв'язування операцій і даних.

Транзакції — це атомарні дії типу «все або нічого», які або завершуються, або припиняються, не впливаючи на решту програми. При одночасному виконанні над спільними даними їх атомарний ефект може бути досягнутий шляхом тимчасового блокування спільних даних для операцій конкретної транзакції, так що дані та операції утворюють тимчасову сутність, яка

розривається після завершення транзакції. Тимчасову сутність можна розглядати як динамічно створений об’єкт, який тимчасово прив’язує дані до операцій транзакції. Під час виконання транзакції її дані доступні тільки локальним операціям, як і локальні дані об'єктів. Транзакції можна розглядати як динамічно створювані часові модулі, які доповнюють текстову модульність програм у просторі модульністю у часовому вимірі.

Транзакції дозволяють структурам спільних даних бути пов’язаними з наборами операцій різних транзакцій на послідовних етапах їх життя. Звичайні об’єкти прив’язують структури даних до операцій у постійному об’єднанні, тоді як у системі транзакцій структури даних можуть бути неоднозначними, маючи безліч різних партнерів. Операції можуть бути однаково безладними, маючи тимчасові зв’язки з багатьма різними структурами даних. Відносна постійність операцій і даних є гнучкою та динамічно визначається.

Чи є транзакції об’єктними? Відповідь – ні, якщо об’єкти розглядаються як фіксовані комбінації даних і операцій. Однак, якщо поняття об’єкта розширюється, щоб включати тимчасові асоціації даних і операцій, тоді транзакції можна розглядати як розширення традиційних об’єктно-орієнтованих понять, щоб включати динамічно створювані об’єкти.

Чи має постійне сховище бути об’єктно-орієнтованим, чи воно має відображати тимчасовий зв’язок операцій із постійними даними. Цілком може бути, що постійне сховище повинно підтримувати слабкі зв’язки між операціями та даними, а також механізми, подібні до транзакцій, для тимчасового зв’язування.

3.2 Формальні обчислювальні моделі

Яка роль математики в моделюванні обчислень? Чи є обчислення просто формою математики, яка вимагає вигнання складних обчислювальних механізмів, таких як призначення та асинхронність? Або обчислювальні моделі слід оцінювати за їхньою здатністю виражати та керувати складністю, щоб

математика стала засобом, а не самоціллю. Об’єктно-орієнтоване програмування є прикладом останнього погляду з багатими та складними математичними моделями, але не має комплексної формальної моделі для повної парадигми.

Об’єкти можуть бути змодельовані автоматами, але теорія автоматів має бути розширена для моделювання систем взаємодіючих об’єктів, особливо за наявності недетермінізму та асинхронності. Типи моделюються за допомогою відношень еквівалентності, алгебр і лямбда-числення. Відношення еквівалентності фіксують класифікаційні властивості типів, алгебри фіксують поведінку типів, а поліморфні типізовані моделі лямбда-числення завершують об’єктно-орієнтовані системи типів. Спадкування моделюється композицією генераторів із незв’язаними (нефіксованими) посиланнями на себе. Взяття фіксованої точки генератора відповідає прив'язці (фіксації) його самовіднесення. Семантика успадкування з фіксованою точкою (денотаційна) розроблена як нове застосування теорії фіксованої точки. Рефлексивні системи досліджують обчислювальні наслідки обробки програм як даних. Вони забезпечують додатковий вимір абстракції (*мета-абстракцію*), яка доповнює абстракцію даних і суперабстракцію. Об’єктно-орієнтовані рефлексивні системи моделюють поведінку класів за допомогою метаоб’єктів, які можуть визначати специфічні для класу рефлексивні дисципліни для налагодження, інтерпретації тощо, із стандартною рефлексивною поведінкою, визначеною в метаметаоб’єкті.

Термін *«автомат»* передбачає негнучкість і безглуздість у застосуванні до людей, але автомати, незважаючи на це, є основним формальним механізмом для моделювання парадигми переходу стану. Вони забезпечують основу для опису машин Тюрінга та цифрових комп’ютерів. Поведінка стимулу/відповіді об’єктів може бути задана автоматами в термінах їх входів I, вихідних даних O, стану S, вихідної функції F і функції переходу стану G (див. Рисунок 3.9):

Автомат = </, O, S, F, G>

Вхідний алфавіт визначає набір можливих подразників, тоді як вихідний алфавіт визначає відповіді. Відповідь на певну вхідну подію залежить від стану. Для заданого входу i та стану s вихід o=F(i,s) визначається функцією виходу F,

тоді як наступний стан s'=G(i,s) визначається функцією переходу стану G.

Вихідна функція F фіксує поведінку стимул-реакція автомата на інтерфейсі, тоді як функція наступного стану G фіксує його внутрішню поведінку. Це відповідає поділу між спостережуваною та внутрішньою поведінкою абстрактного типу даних.

Теорія скінченних автоматів була багатою областю досліджень наприкінці 1950-х і 1960-х років. Він включав як синтез автоматів із послідовних схем, так і алгебраїчну теорію автоматів. Було показано, що будь-який автомат можна побудувати як композицію логічних елементів *nand* або *and/not* , або алгебраїчну композицію простих примітивних напівгруп. Але ці результати передбачали синхронну композицію з виходом на одному кроці часу, доступним як вхід на наступному кроці часу. Об’єкти підпорядковані більш складним і ще не повністю вивченим протоколам зв’язку та законам композиції.

Об’єкт зі станом s і операціями (методами) fl,f2,...,fN можна розглядати як автомат, який, отримавши вхід fi з аргументом x, виконує такі дії:

він повертає результат fi(x, s)

він виконує перехід стану в новий стан gi(x, s)

Об'єкти можна розглядати як автомати, вхідний алфавіт яких складається з пар вхідних аргументів (fi,x). Кожен вхід може викликати як вихід, так і перехід стану. На рисунку 3.9 зображено об’єкт зі станом s, операціями fl, f2,... , fN і функціями переходу стану gl, g2 ..... gN. Об'єкти ген

зазвичай не розділяйте функції виведення та наступного стану, причому кожна операція fi має похідну функцію переходу стану gi.

Об'єктні автомати мають структуру скінченних автоматів [LP], але їхні стани можуть ставати як завгодно великими, як у випадку необмежених буферів або стеків. Послідовність пар операція-аргумент (f,x) автомата визначає історію його обчислень і іноді називається «слідом». Слід грає роль вхідної стрічки традиційного автомата.

Послідовні процеси Хоара [Но3], Мілнера [Мі] і Хенессі [Не] моделюють окремі процеси за допомогою автоматів і підтримують композицію паралельних

процесів. Приклад мови процесів Henessy EPL, який відображає моделі процесів Хоара і Мілнера, визначає процеси в термінах алфавіту подій і операції *становлення* (—>).

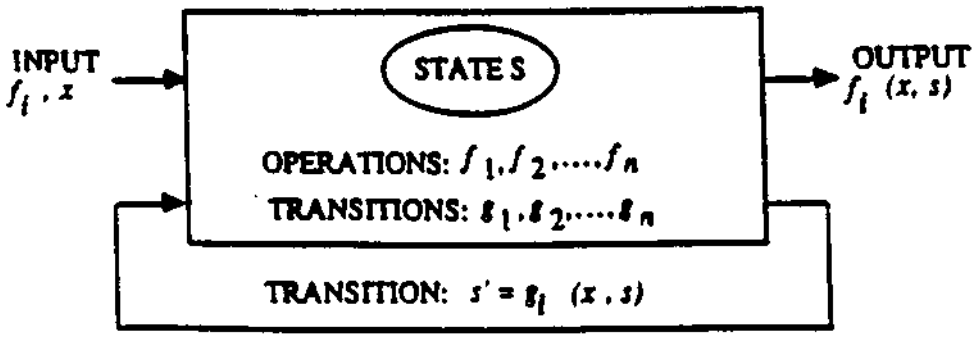


Рисунок 3.9 - Об'єкти як автомати

"P —> aQ» означає: «процес P, на який діє подія a, стає Q». «P —> aP» означає, що на P може впливати послідовність a -подій, а «P —> abP» означає, що на P може впливати послідовність подій «a, за якими йде b». Вибір P між подіями a і b позначається «P —> aX + bY». Процес «P — » acP + bdP» допускає послідовність «a, за якою йде b». Події c" або "b, за якими йде d". Параметризовані процеси можуть вказати вибір серед набору подій зі схожими властивостями *стати* : P(x:B) —> Q(x) означає, що P, діючи на подію x у наборі B, стає Q(x).

Послідовність дій процесу під час обчислення називається *трасою.* Поведінка процесу може частково характеризуватися наборами можливих слідів. Але траси не є повними специфікаціями поведінки: «P —> a(b+c)X» і «Q —> abX+acX» мають однаковий набір трас, але різну поведінку, оскільки після дії a P може виконати b або c, тоді як Q може виконати лише b. Поведінково еквівалентні процеси повинні продовжувати мати той самий набір слідів після виконання будь-якої дозволеної початкової послідовності дій. Еквівалентність слідів — це *відношення конгруенції,* яке зберігається виконанням дій еквівалентних автоматів.

Вихідні події позначаються простими цифрами, де a' означає вихідну подію, що відповідає вхідній події a. Паралельні процеси спілкуються шляхом

зустрічі між входом і відповідним виходом. Таким чином, «R —> aR» і «S — » a'S» можуть постійно спілкуватися за допомогою S, надсилаючи a-повідомлення R.

Композиція процесу P і Q позначається «P | Q» і дозволяє P і Q виконуватися паралельно (паралельно реагувати на події, які керують їх виконанням). Якщо «X — » acX» і «Y — » ac'Y», тоді «XY = X | Y» дозволить a -події X і b-події Y виконуватися паралельно, а потім потребуватиме зустрічі між c-подією X і c'-подією Y.

Приховування інформації внутрішніх подій може бути реалізовано шляхом *обмеження.* XY/{c) обмежує XY так, що c є внутрішнім і тільки a і b видимі для зовнішнього світу. Обмеження робить внутрішні дії процесу *недоступними для спостереження.* Таким чином, R|S/{ a) є *аутичним* процесом, який не має зв’язку із зовнішнім світом, який назавжди виконує неспостережувану подію a.

Обмеження — це *паралельний оператор абстракції даних* , який створює непослідовні процеси зі складнішою внутрішньою структурою, ніж традиційні автомати, фіксуючи властивості агентів реального світу з непослідовною внутрішньою поведінкою. Такі агенти більше не можуть моделюватися стандартною теорією автоматів. Концепція *спостережуваної поведінки* для таких агентів ускладнюється неспостережуваними внутрішніми діями, які можуть викликати непередбачувані зміни поведінки інтерфейсу. Визначення *спостережуваної еквівалентності [Mil]* Мілнера в термінах бісимуляції (двостороннє моделювання) забезпечує основу для точного визначення спостережуваності.

Наведені вище примітиви дозволяють побудувати складні процеси з цікавою поведінкою та гарними алгебраїчними властивостями [He]. Але асинхронність і недетермінізм призводять до математичних ускладнень у визначенні поведінки великих (реальних) програм. Крім того, протокол зв’язку двостороннього іменування не фіксує поведінку процесів клієнт/сервер, і існують інші аспекти, в яких формалізм є занадто простим, щоб безпосередньо

моделювати паралельність у реальному світі.

Композиція об’єктів реального світу може включати численні взаємозалежні взаємозв’язки (сіамські близнюки) та реструктуризацію інтерфейсів (за допомогою *необмежених* операторів) для усунення внутрішніх комунікаційних бар’єрів. Поведінка складних об’єктів (корпорацій, багатоособових дослідницьких груп, людського тіла) не може бути легко або одноманітно визначена в термінах поведінки їхніх компонентів.

Поведінку суспільства не можна легко передбачити на основі поведінки окремих осіб, які його складають [Мін]. Проблеми композиції декларативного об’єкта та процесу набагато складніші, ніж проблеми композиції імперативної функції. Можливо, занадто багато очікувати, що технологія компонентів програмного забезпечення вирішить ці проблеми, але ми, тим не менш, повинні усвідомлювати великий розрив між моделями та реальністю.

Зробивши спрощені припущення, які усувають асинхронність, можна розробити математично придатну модель паралельних обчислень. Синхронні системи складаються з поточних виконуваних модулів, які синхронно оновлюються на кожному кроці обчислення. Рисунку 3.10 ілюструє одночасне виконання автоматів Al, A2,..., AN у станах si, s2,...,sN, які можуть бути синхронно оновлені до станів si', s2', ...,sN' за один крок обчислення. Стан глобальної системи є перехресним продуктом станів компонентів.

Наскільки корисними є синхронні системи як практичний інструмент для моделювання паралельності? Коли час обчислення в модулі або час зв’язку між модулями перевищує інтервал синхронного обчислення, тоді синхронні системи нереалістичні. Але коли час обчислень і зв’язку малий порівняно з інтервалом між кроками обчислень, то синхронні системи є хорошим наближенням. Це стосується багатьох реактивних систем реального часу, таких як системи керування ліфтами, хімічними заводами та електростанціями. Для таких систем адекватною та корисною є модель синхронно працюючого суперавтомату з паралельно виконуючими синхронними компонентами.

Синхронні системи припускають, що обчислення та зв’язок є миттєвими,

що відбуваються в момент переходу стану. Оскільки вихідні дані певного модуля миттєво доступні в усій системі, широкомовна передача є більш прийнятною моделлю зв'язку, ніж традиційна передача повідомлень. Кожен модуль може на кожному кроці обчислення отримувати миттєво передані повідомлення від будь-якого іншого модуля. Ми можемо думати про комунікаційне середовище як про ефір, у якому повідомлення подорожують, як світло, у всіх напрямках із фактично нескінченною швидкістю, і на них може впливати будь-який спостерігач, налаштований на його довжину хвилі.

Взаємозалежність між модулями визначається глобальною функцією переходу стану. Міжмодульний зв'язок може бути абсолютно різним для різних станів і є більш гнучким, ніж для статично взаємопов'язаних мереж. Синхронні системи вперше були розроблені в контексті Esterel Беррі та Гонтьє [BG], а також є прикладами Statecharts [Har].

Типи породжують глибокі філософські та теоретичні питання не лише для мов програмування, а й для математики та біології [We2]. Який зв'язок між типами та значеннями? Чи є значення фундаментальними, а типи – просто похідними класами значень, чи типи є базовими концептуальними сутностями та значеннями, доступними лише через фільтр інтерпретації типу? Філософська дискусія між реалістами, які вірять у примат цінностей, та ідеалістами, які вірять у ідеали Платона, має відповідник у математиці та фізиці [We2].

Властивості типів дуже відрізняються від властивостей значень. Значення перетворюються за допомогою правил обчислення (правил редукції), тоді як типи моделюють структуру або поведінку, які безпосередньо не пов'язані з обчисленням результату. В ідеалі ми хотіли б інтегрувати типи та значення в єдину безшовну систему, але цей ідеал не був реалізований на практиці і може бути теоретично неможливим.

Вирази типів генеруються з базових типів конструкторами типів:

тип —> основний тип | побудованого типу

основного типу — » Між | Real \ Bool | Чар ...

сконструйований тип —> тип масиву \ тип запису \ тип функції ...

тип масиву —> масив [розмір] типу елемента

тип запису —> запис[{ідент:тип}+]

тип функції —> (тип домену —> тип діапазону)

Нас цікавлять наступні запитання щодо виразів типу та пов'язаних типів:

Еквівалентність: *коли два вирази типу TE1 і TE2 еквівалентні?*

Поліморфізм: *які відносини між виразами типу TE1 і TE2 можуть бути виражені?* **Семантика:** *задано вираз типу TE, яка семантика асоційованого типу T?* **Поведінка:** *Яка поведінка типу T?*

Модифікація поведінки: *як T2 може змінити поведінку T1?*

Перевірка типу: *чи має змінна x тип T?*

Висновок типу: *що можна зробити висновок про тип виразу з його контексту?*

Різні питання про тип ведуть до різних видів математичних теорій. Питання еквівалентності типу та поліморфізму можуть бути змодельовані за допомогою відношень еквівалентності, питання поведінки типу за допомогою алгебр, а питання перевірки типу та виведення типу за допомогою типізованого поліморфного лямбда-числення.

Кожна з наступних додаткових моделей типу має різні цілі:

моделі класифікації в термінах відносин еквівалентності *моделі поведінки* в термінах алгебр

повні мовні моделі в термінах поліморфного типізованого лямбда-числення

Відношення еквівалентності є основним класифікаційним механізмом математики. Моделюючи типи як відношення еквівалентності, ми фіксуємо їхні класифікаційні властивості. _

Системи простих типів класифікують значення універсального набору значень у непересічні класи еквівалентності (див. Рисунок 3.10). Кожне значення належить тільки до одного типу. Але може бути нескінченна кількість ієрархів таких типів, як «(Int — » Int)» та «((Int —> Int) —)• Int)» тощо.

Відношення еквівалентності є надто жорстким механізмом класифікації,

оскільки вони класифікують значення на непересічні класи. Ми послаблюємо цю жорсткість, спочатку дозволяючи часткову класифікацію, у якій деякі значення залишаються некласифікованими, а потім дозволяючи класи значень, що перекриваються.

Один тип може бути змодельований відношенням часткової еквівалентності (PER), формально визначеним як відношення еквівалентності, яке є симетричним і транзитивним, але не рефлексивним (Рисунок 3.11). Рефлексивність справедлива лише для значень типу, які утворюють острови типоеквівалентних значень у морі нерефлексивних значень інших типів. Системи поліморфних типів дозволяють значенням мати більше одного типу, що відповідає ситуаціям, коли значення підлягають різним правилам обчислення в різних контекстах використання.

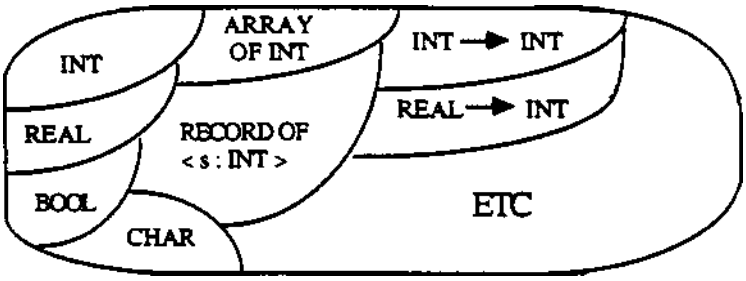


Рисунок 3.10 - Типи як розділи універсального набору значень

Наприклад, людина поводитиметься по-різному в сім'ї та на роботі, програма буде поводитися по-різному в середовищі під час компіляції та під час виконання, ціле число може подвоюватися як дійсне або комплексне число, а Toyota може бути типу Car and Vehicle.

Системи поліморфного типу можуть бути змодельовані "відношеннями сумісності", які є рефлексивними та симетричними, але не транзитивними. Розмовляти спільною мовою - це відношення сумісності. Якщо А розмовляє англійською та французькою, В розмовляє французькою та німецькою, а С розмовляє німецькою та іспанською, то А сумісний з В, В сумісний з С, але А не сумісний з С.

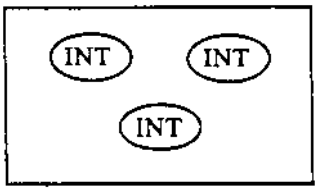


Рисунок 3.11 - Острови, що визначаються відношеннями часткової еквівалентності

Відношення сумісності визначають *покриття* своєї області шляхом перекриття класів сумісності. Наприклад, мови охоплюють групу людей шляхом накладання класів, так що кількість класів, до яких належить особа, є кількістю мов, якими розмовляють. На рисунку 3.11 цілі, дійсні та комплексні числа є виродженим покриттям, у якому комплексні числа включають дійсні числа, які, у свою чергу, включають цілі числа.

Класифікація на непересічні класи фіксується відношеннями еквівалентності, класифікація на класи, що накладаються, завдяки послаблюючій транзитивності, а зосередження на одному класі - послаблюючій рефлексивності. Досить примітно, що ці прості абстрактні умови забезпечують таку велику гнучкість у визначенні практичних систем класифікації.

Elim ina tin g рефлексивність спричиняє належність кожного значення щонайбільше до одного класу, усунення транзитивності спричиняє належність кожного значення принаймні до одного класу, а вимога рефлексивності, симетрії та транзитивності спричиняє належність цінностей до точно одного класу.

Відношення еквівалентності визначають, коли два значення належать до одного типу, але нічого більше не говорять про властивості типу. Щоб визначити поведінку типу, нам потрібна сильніша структура елементів набору. Алгебри накладають структуру на множини за допомогою операцій і забезпечують основу для моделювання класів і абстрактних типів даних.

Дані та пов’язані з ними операції класу або типу мови програмування можуть бути математично змодельовані за допомогою алгебри:

Алгебра = <набір значень; операції>

Операції мають *арність,* яка визначається кількістю їх операндів. Набір S

разом з операціями та їх арністю називається *сигнатурою.* Операції з нулем, одним і двома операндами називаються нульовими, унарними та двійковими операціями.

Алгебра цілих чисел при додаванні та множенні має набір I цілих чисел, дві двійкові операції "+,*" і дві нульові операції "0,1":

Цілі числа = <1; +,,0,1>*

Щоб завершити визначення алгебри, ми повинні визначити, як її операції перетворюють її елементи. Це може бути визначено правилами, алгоритмами або апаратним забезпеченням для застосування операцій до значень. Слабша специфікація властивостей алгебраїчних операцій, яка обмежує їх поведінку без повного визначення, може бути дана аксіомами:

i+j = j+i комутативність (i+j)+k= i+(j+k) асоціативність i(j+k) = i*j + i*k дистрибутивність i*l = i мультиплікативна тотожність i: i*0 = 0 мультиплікативний нуль*

Змінні в цих аксіомах мають неявні універсальні квантори: V i,j: i+j = j+i

Набір I називається «сортуванням», а алгебру цілих чисел називають односортованою алгеброю, оскільки всі операції приймають аргументи та повертають значення лише одного сортування.

Об’єктно-орієнтовані операції зазвичай вимагають кількох різних типів аргументів. Таким чином, тип моделюється багатосортною алгеброю, сигнатурою якої є набір сортів S і набір функцій fi з типом Ti:

Many-sorted-algebra = <Набір сортувань: fl:Tl,f2:T2 ,... >

Наприклад, тип стека — це багатосортова алгебра з чотирма сортуваннями та трьома операціями:

Набір сортувань: (Int, List(Int), Bool, Void)

*Набір операцій: push: List(Int) * Int —> Void pop: List(Int) — » Int empty: List(Int) — » Bool*

Поведінка операцій може бути частково визначена аксіомами:

pop(push(x, стек)) = x порожній(push(x, стек)) = false

Ці аксіоми накладають обмеження на операції зі стеком, які є

обґрунтованими в тому сенсі, що вони узгоджуються з фактичною поведінкою стеків. Знайти аксіоми, які є *повними* в тому сенсі, що вони повністю визначають поведінку стека, складніше, частково тому, що поведінка стека є неформальним поняттям. Головна мета алгебр - охопити семантику поведінки за допомогою набору аксіом із суто синтаксичними властивостями.

Алгебри моделюють поведінку операцій за допомогою аксіом рівняння, а не шляхом присвоєння значень змінним. Вони можуть фіксувати поведінку функціональних об'єктів і таких об'єктів, як стеки, з неруйнівними операціями push і pop, але не імперативних об'єктів із оновлюваним станом.

Алгебри були розроблені для моделювання математичних об'єктів, таких як цілі та дійсні числа. Для моделювання класів потрібне розширення до багатосортних алгебр. Подальше розширення до *порядково-сортованих алгебр* [GM] потрібне для охоплення зв'язків упорядкування між сортуваннями, які виникають у підтипах і спадкуванні. Алгебраїчні моделі підтипу також обговорюються в [BW], де порівнюються поняття підтипу, визначені в термінах підмножин, ізоморфного вкладення та обмежень предикатів. Існує мало традиційної математики щодо відношень другого порядку між алгебрами, необхідних для охоплення поняття успадкування та інших форм поліморфізму типів.

Обчислення моделює семантичні властивості реального світу за допомогою чисто синтаксичних символьних перетворень. Математика також має формальні мови з чисто синтаксичними перетвореннями символів, які спрямовані на моделювання семантичних властивостей функціональних областей застосування. Математичне поняття доказу у формальній системі відображає поняття обчислення: теорема, що доводиться, відповідає поняттю обчислювальної величини.

Канонічною формальною системою в математичній логіці є числення предикатів, яке забезпечує універсальну логічну структуру для предметно-спеціальної формалізації областей застосування. Навпаки, алгебри моделюють спеціалізовані типи поведінки, а аксіоми рівнянь для конкретної алгебри можна

розглядати як синтаксичні наближення до семантичної поведінки, втіленої алгеброю.

Що спільного між обчислювальною та математичною моделями? Обидва мають на меті охопити семантичні поведінкові властивості за допомогою чисто синтаксичних перетворень (обчислення та доказ). Обидва визначаються мовами для перетворення синтаксичних символів, які спрямовані на захоплення незалежно визначеної семантичної поведінки. У математиці розрізняють *теорію доказів,* що стосується методів доказів, і *теорію моделей,* що стосується внутрішньої семантики області, що моделюється. Для обчислень теорія доказів відповідає операційній семантиці, тоді як теорія моделі відповідає денотаційній семантиці. Основна мета семантики полягає в тому, щоб продемонструвати адекватність операційних, обчислювальних і теоретико-доказових синтаксичних формалізмів для охоплення семантики домену, що моделюється.

Теорія моделей визначається в [СК] як «алгебра + логіка». Математична модель відображає твердження щодо алгебраїчних виразів у значення істинності. Істина - семантичне поняття, що відрізняється від синтаксичного поняття доказовості. *Теорія доказів* стурбована властивостями доказів, тоді як теорія моделей стурбована тим, чи є твердження, доведені у формальній системі, істинними в моделях цієї системи.

Враховуючи формальну систему, яка визначає доказовість, і модель, яка визначає істину, формальну систему називають *надійною* (щодо моделі), якщо все, що можна довести, є істинним, і *повною,* якщо все істинне є доказовим. Алгебра є надійною моделлю поведінки, якщо властивості операцій, які виводяться з аксіом, узгоджуються з поведінкою, і є повною моделлю поведінки, якщо всі її властивості виводяться з аксіом. Екваціональні аксіоми алгебри фіксують поведінку моделі, якщо вони є обґрунтованими та повними для цієї моделі. Таким чином, модельна теорія встановлює відношення між поведінкою, визначеною формальними системами, і поведінкою, визначеною незалежним поняттям істини.

Яке значення має це обговорення для встановлення того, що програма

фіксує бажану неформальну поведінку? На практиці ми не можемо визначити надійність і повноту алгебр для неформальної поведінки. Найкраще, що ми можемо зробити, це визначити обґрунтованість і повноту певної форми малізації неформальної поведінки. Це залишає відкритим питання про те, як формалізувати неформальну поведінку та як можна продемонструвати адекватність формалізації. Ми ніколи не можемо повністю встановити валідність формальних систем як моделей неформальних концепцій, тому що це було б рівнозначно їх формалізації. Але математичні моделі фіксують неформальні поняття більш безпосередньо, ніж системи аксіом. Існують дійсні аналогії між неформальним і математичним моделюванням, які необхідно краще зрозуміти.

Лямбда-числення [Ba] історично було розроблено як формалізм для обчислень зі значеннями, а пізніше модифіковано, щоб включити типи. Його простота дозволяє нам чітко продемонструвати взаємодію типів і значень у обчислювальному формалізмі. Поліморфне типізоване лямбда-числення моделює поліморфні типи в контексті зменшення значення лямбда-числення.

Синтаксис лямбда-виразів можна вказати так:

вираз —> змінна \ функція \ комбінована функція —tfun(змінна). вираз - абстракція поєднання — » вираз (вираз) -додаток

Двома основними семантичними поняттями є *додаток* і *абстракція.* Програма просто застосовує один вираз до іншого та викликає суттєве обчислення (зменшення), якщо перший вираз є специфікацією функції у формі "fun(x). E".

Абстракція перетворює вираз E у функцію "fun(x). E", параметризуючи E змінною x, яка може зустрічатися в тілі E. Абстракція тут означає *функціональну абстракцію* та служить для вказівки на те, що x є змінною частиною виразу *E(x),* тоді як його решта структури залишається незмінною. Якщо *E = f(x),* то функціональна абстракція параметризує аргумент x і залишає функцію f інваріантною. За аналогією ми могли б визначити оператор абстракції даних *data(f).f(x)* , який параметризує функціональну частину та залишає інваріантною частину даних, щоб різні функції могли застосовуватися до заданих даних. Тоді

ми могли б охарактеризувати абстракції функцій як такі, що мають форму *data(f). весело (x). f(x)* і абстракції об'єктів як такі, що мають форму *fun(x). дані(f).f(x).* Тобто функціональні абстракції надають параметри даних незмінним функціональним тілам , тоді як абстракції об'єктів надають функції (повідомлення), які потрібно інтерпретувати в об'єктному середовищі. Однак, оскільки лямбда-числення формально не розрізняє функції та дані, лише одне

оператора абстракції *fun* (або *лямбда)* достатньо.

Функції наступника та двічі визначаються такими лямбда-виразами:

succ = fun(x). x+1 двічі

=fun(f).fun(y)f(f(y))

Основним правилом обчислення в лямбда-численні є правило редукції, яке вказує, що комбінація у формі "fun(x). M (N)" обчислюється шляхом заміни N на екземпляри x у M, як у наступній послідовності скорочень:

двічі(succ)(3) —¥ fun(f).fun(x).f(f(x)) (succ) (3) -> fun(x). succ(succ(x)) (3) — > succ(succ(3)) — » 5

Це обчислення застосовує послідовність правил скорочення до початкового лямбда-виразу, доки не буде отримано незвідний вираз (званий значенням), який не можна далі скоротити.

Типізоване лямбда-числення пов'язує типи зі зв'язаними змінними так, що синтаксис функцій змінюється таким чином:

*функція —*fun(змінна: тип), вираз*

Типізована функція наступника має такий вигляд:

succ = jun(x: Int). x+1

Це призводить до заборони застосування succ до аргументів не типу Int і передбачає перевірку типу для визначення сумісності типів. Властивість нетипового числення, що будь-який вираз застосовний до будь-якого іншого виразу, не відповідає дійсності в типізованому численні. Типізоване лямбда-числення забезпечує безпеку, гарантуючи, що перанди комбінацій сумісні з їх операторами, але за рахунок розширення чистих правил заміни правилами перевірки типу, які не мають відношення до правил обчислень.

Типізоване лямбда-числення не має механізму для вираження структурної подібності типів. Для типів Int і Real необхідно окремо вказати функцію double:

twoint =fun(f: Int -» Int).fun(x: Int).f(f(x)) tworeal =fun(f: Real -+ Real). весело (x: Справжній). f(f(x))

Така структурна подібність може бути виражена виразами поліморфного типу: *twoall = all[t] fun(f: t —> t).fun(x: t).f(f(x))*

Цей вираз має такий поліморфний (універсальний) тип:

V *m. ((t ->t)*t)-+t*

Тобто для всіх функцій типу (t - » t) і аргументів t повертається результат типу t.

Twiceall фіксує єдину структуру подвійного застосування для всіх типів t. Він може бути спеціалізований, надаючи параметр типу:

Двічі все (Int) = два рази Двічі все (Реальний) =

двічі дійсне

Якщо twoall надається з параметром типу Int, за яким іде функціональний параметр succ, а потім цілочисельний параметр 3, він застосовує succ двічі до 3, даючи 5:

двічівсе(Int)(succ)(3) —>5

Універсальні типи охоплюють певний вид поліморфізму: той, що пов’язаний з параметрами типу загальних функцій. Мова Fun [CW] розширює типізоване лямбда-числення трьома типами поліморфних типів: універсальні, екзистенціальні та обмежені типи:

універсальний тип -+ \t змінна, тип екзистенціальний тип - незмінний, тип обмежений тип - i V *variable < type, type*

Екзистенціальні типи фіксують приховану інформацію та абстрактні типи даних, тоді як обмежені типи фіксують успадкування. Таким чином Fun моделює системи типів, які виникають в об'єктно-орієнтованому програмуванні.

Перевірка типу та визначення типу

Перевірка типу може бути формально визначена набором правил (аксіом),

таких як наступне правило для застосування функції:

враховуючи, що: f має тип A -+ B і x має тип A, виведіть, що: f(x) є правильним типом і має тип B

Це правило одночасно є правилом перевірки типу для демонстрації правильності типу f(x) і правилом виведення типу для виведення типу f(x) з типу f і типу x. Між перевіркою типу та виведенням типу існує тісний зв'язок, оскільки перевірка типу потребує виведення типу, щоб продемонструвати правильність. Наприклад, щоб продемонструвати коректність типу "g(f(x))", де g має тип "B — » C", ми повинні використовувати висновок, що f(x) має тип B.

Правила перевірки типів визначають семантику під час компіляції для типів, яка має мало спільного з поведінкою під час виконання. Типізоване лямбда-числення має два абсолютно різні набори правил, один для перевірки типу виразів, а інший для обчислення зі значеннями. Це відповідає різниці між компіляцією та інтерпретацією програм, і тому цілком природно. Однак - отриманий формалізм є гібридом двох різних типів виразів із гібридними правилами обчислення, який більше не володіє первозданною простотою чистого лямбда-числення.

Системи виведення типу дозволяють програмісту писати неявно типізовані вирази [BH], явний тип яких автоматично визначається системою виведення типу. Мови з визначенням типу, такі як ML, підтримують такий стиль комп'ютерного діалогу:

введення від особи: 3 + 4 Відповідь комп'ютера: 7: Міжн

Тут комп'ютер робить висновок, що 3 і 4 мають тип Int, обчислює результат і інформує користувача, що результат має тип Int. Коли особа вказує функцію, система визначення типу визначає тип функції:

*введення від особи: succ = fim(x). Відповідь комп'ютера x+1: Int — » Int введення особою: двічівсе = fun(f). весело (x). f(f(x)) відповідь комп'ютера: Xt t. ((t — » t) * t) — » t*

Щоб зробити висновок про те, що *twoall* є універсальним поліморфним типом, система виведення типу ML повинна визначити найзагальніший тип, який

відповідає всім контекстам використання, в яких може мати місце підвираз. Якщо підвираз не обмежує контексти, в яких його можна використовувати, системі, можливо, доведеться винайти нові найзагальніші типи. Ідея типів як *найзагальніших контекстів* радикально відрізняється від ідей типів як поведінки: відображення синтаксичних контекстів у семантичні поведінки не завжди є простим.

Ідея виведення типу була розроблена у зв'язку з ML [ML]. Універсальні поліморфні типи необхідні в ML не тільки тому, що вони корисні, але також тому, що функції, такі як twoall, які не розкривають контекст їх використання, не можуть бути типізовані інакше.

У системах виведення типу тип виразу типу - це набір усіх контекстів, у яких вираз може мати значення. Ідея типу як набору контекстів інтерпретації дуже відрізняється від точки зору, що тип є поведінкою або класом еквівалентності значень. Чітке розуміння інтуїції, що лежить в основі кожного з цих понять типу, допомагає нам краще зрозуміти пов’язані математичні моделі.

Мови з виведенням типу дозволяють користувачам їсти свій торт і також його; користування перевагами безпеки типу без тягаря явних декларацій типу. Але неявно типізовані значення мають дещо відмінну семантику від нетипізованих значень. Вони вводять найзагальніші поліморфні типи, окрім тих, які необхідні для опису предметної області. Скромний набір типів, обраних для моделювання поведінки, може призвести до ідіосинкратичних контекстуальних типів, які не відповідають безпосередньо поведінці. Канеллакіс, Мейрсон і Мітчелл [KMM] показали, що обчислювальна складність виведення типу є в принципі експоненціальною, хоча вирази, які викликають виведення експоненціального типу, навряд чи виникнуть на практиці.

Що математична семантика може внести в дискусію між прихильниками типізованих і нетипізованих мов? Теорія математичних моделей встановлює зв’язок між програмами та їхніми позначеннями, який відповідає їхній обчислювальній семантиці. Типовий висновок відображає нетипові в явно типізовані вирази. Рисунок 3.12 стисло виражає співвідношення між

нетипізованими і типізованими моделями. Він забезпечує структуру для вивчення компромісів між a) безпосереднім виконанням нетипової програми та b) перетворенням її на типізовану програму та виконанням типізованої програми.

Враховуючи мову у верхньому лівому куті рисунку 3.12, верхня горизонтальна лінія представляє моделі нетипізованої мови, ліва вертикальна лінія представляє систему виведення типу з нетипізованої мови на типізовану, а нижня горизонтальна лінія представляє модель типізованої мови. Права вертикальна лінія представляє математичні моделі від нетипізованих до типізованих позначень. Якщо *"type(untyped-denotation(program)) = typed-denotation(type-inference(program))"* для всіх програм, тоді діаграма змінюється. Однак деякі запропоновані моделі не комутують, і тому рисунку 3.12 не припускає, що введення нетипових позначень завжди буде еквівалентним виведенню типу з наступним прийняттям введеного позначення.

Перші моделі лямбда-числення були нетиповими, забезпечуючи вільні від парадоксів позначення для нетипізованих лямбда-виразів у ґратчасто-

структурованих областях неперервних функцій [Ba, Sc]

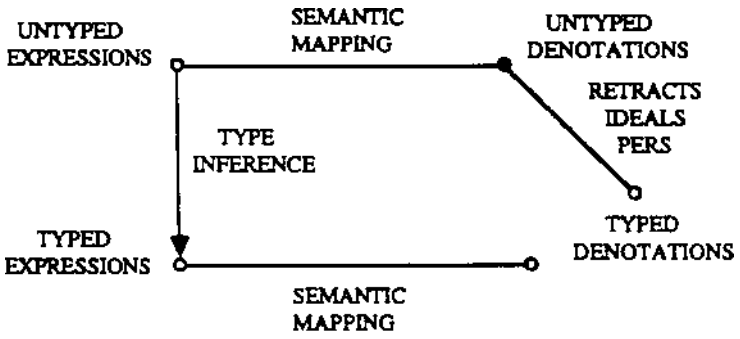


Рисунок 3.12 - Зв’язок між нетипізованим і типізованим лямбда-численням

зв'язок між нетиповими і типізованими моделями розглянуто в [BH] і більш детально в [Mey, BMM]. Незважаючи на те, що нетипізовані моделі історично передували типізованим моделям, семантика нетипізованих виразів точніше фіксується шляхом їх першого зіставлення з типізованими виразами, ніж нетипізованими позначеннями. Ми можемо виразити семантику простіше та

безпечніше, спочатку відобразивши нетипові у типізовані вирази та обмеживши наше денотаційне відображення виключно типізованими виразами. Прихильники типізованих мов дотримуються наступного принципу:

Принцип типізації: прив'язка виразів до типу має передувати прив'язці їх до позначення

Математичні формалізми, включаючи мови програмування, безпечніші, якщо вони мають явне поняття типу. Вони мають простішу операційну та денотаційну семантику, ніж їхні нетипізовані відповідники, тому що семантику потрібно надавати лише для виразів, сумісних за типом. *Введення перед оцінкою* - це стратегія «розділяй і володарюй», яка поділяє всесвіт значень на введені категорії, щоб легше перемогти його за допомогою обчислень.

Контраргументи про те, що типізація є додатковим багажем, стороннім для прямих поведінкових властивостей цінностей, що типізація упереджує і, отже, обмежує контексти, в яких значення можуть використовуватися d, і що нетипізовані рефлексивні системи є набагато простішими, ніж відповідні типізовані рефлексивні системи, наведені в іншому місці цієї статті.

Повне розуміння рисунка 3.12 вимагає володіння теорією моделей для нетипового і типізованого лямбда-числення, алгоритмів виведення типу для мов, таких як ML, і механізмів моделювання типів, таких як відкликання, відношення часткової еквівалентності та ідеали [We2]. Проте цілі теорії нетипізованих і типізованих моделей, а також відображення виведення типів з нетипізованих на типізовані мови можуть бути зрозумілі навіть нематематикам.

У чому суть спадкування? Що спільного між різними видами успадкування, розглянутими в розділі? Яка різниця між успадкуванням суперкласу та простим викликом його операцій? Спадкування - це механізм спільного використання та повторного використання поведінки. Він відрізняється від інших механізмів спільного використання поведінки відкладеним зв'язуванням самопосилання, так що суперкласи можуть об'єднати свою ідентичність із підкласами, які їх успадковують.

Успадковані атрибути більш істотно є частиною підкласу або об'єкта,

який їх успадковує, ніж атрибути, які просто використовуються або викликаються, так само як кольори очей є більш істотною частиною людей, ніж автомобіль, яким вони їздять, або будинок, в якому вони живуть. Злиття ідентичності усвідомлюється пізно (час виконання) прив'язка самопосилань до об'єкта, від імені якого виконується операція. На рисунку 3.13 показано об’єкт X, що належить до класу M з батьківським класом P. Пізнє зв’язування - самопосилання в P дозволяє йому вважати ідентичність кожного модифікуючого підкласу M, який успадковує його, зберігаючи свою текстову незалежність. Ідентифікація як M, так і P пов’язана з об’єктом X у той момент, коли X запитує M виконати повідомлення від його імені.

Прив’язка часу виконання «self» у той час, коли об’єкт X відповідає на повідомлення, може бути реалізована простою ініціалізацією вказівника на базовий клас, у цьому випадку клас M. Усі посилання на себе під час виконання цієї операції, незалежно від того, з якого суперкласу, інтерпретуються як посилання на цей вказівник. Це призведе до того, що всі посилання на себе як у P, так і в M починають шукати в M операцію, яку потрібно виконати, і шукати в P якщо операція не знайдена в M.

Цей простий механізм реалізації для прив’язки часу виконання самопосилання має гарну математичну модель з точки зору теорії фіксованої точки. Використання фіксованих точок у рекурсивних функціях видно з наступного визначення факторіальної функції:

*fact =fun(n). ifn = 1 then 1 else n * fact(nl)*

Слідуючи Куку [Co], ми перетворюємо праву частину на нерекурсивну функцію, розглядаючи її самопосилання як зв’язану змінну, значення якої надається пізніше.

*ФАКТ = fim(self).fun(n). ifn = 1 then 1 else n * self(nl)*

ФАКТ називають генератором факту. Генератори зв'язують вільне самопосилання рекурсивної функції, щоб вона стала параметром генератора. FACT - це функціонал, який визначає функцію від цілих чисел до цілих, коли йому надається значення її формального параметра self. Коли FACT

застосовується до факту, виходить факторіальна функція.

ФАКТ(факт) = факт

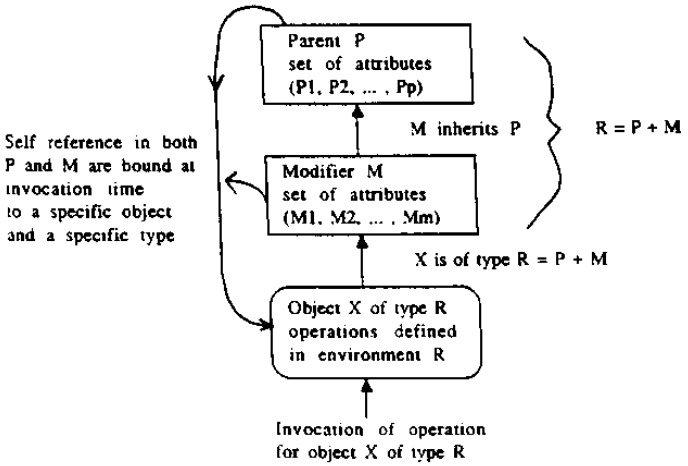


Рисунок 3.13 - Прив'язка часу виконання до самопосилань

Фіксованою точкою функції f є будь-який x такий, що f(x) = x. Факторіальна функція є фіксованою точкою свого генератора FACT, і ми можемо написати "FEX(FACT) = fact", де FIX є операцією взяття (найменшої) фіксованої точки її аргументу.

Відношення між нефіксованим генератором FACT і його фіксованою точкою показано на рисунку 3.13. Нефіксований генератор має висяче посилання, тоді як фіксована точка закріплює самопосилання в самій сутності.

Загалом, будь-яка рекурсивна функція виду: "f = body з екземплярами f' може бути перетворена на нерекурсивний генератор: "F = fun(self). body-with-instances-of-self" з властивістю : "F(f) = f".

Наведене вище обговорення генераторів для рекурсивних функцій можна узагальнити на рекурсивні класи. Нехай P - рекурсивний клас, що містить входження self як вільної змінної. За аналогією з рекурсивними функціями, генератор GENP для P можна визначити як "GENP = fun(self). P".

Як і у випадку рекурсивної функції, P є фіксованою точкою генератора GENP:

GENP(P) = P FIX(GENP) = P

Фіксація GENP прив’язує його висяче самопосилання до класу P (див.

Рисунок 3.14).

Генератори представляють рекурсивні сутності, у яких самопосилання є параметром. Взяття фіксованої точки генератора відповідає прив'язці його самопосилання, таким чином надаючи самопосиланню певне позначення. GENP, як і FACT, є генератором із висячим посиланням на себе. Говорячи про фіксовану точку GENP, ми закріплюємо само посилання.

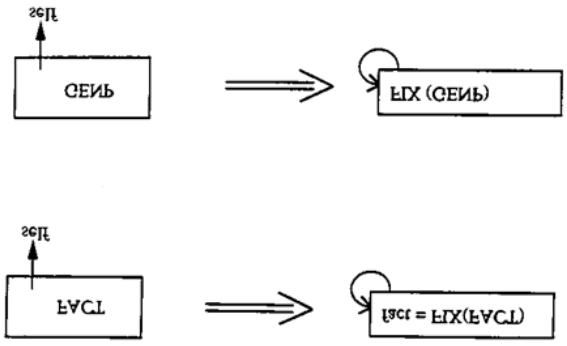


Рисунок 3.14 - Генератори функцій та їхні нерухомі точки

Затримки успадкування беруть фіксовану точку генераторів класів, щоб дозволити композицію з підкласами. Нехай M - клас, який успадковує P, і нехай «GENM = fun(self). M» - генератор M. Успадкування P від M можна визначити в термінах складу генераторів P і M:

GENPM = СКЛАДИ(GENP, GENM) =fun(self). compose(GENP(self), GENM(self)) Inherits(P, M) = FIX(GENPM)

GENPM складається з GENP(self) і GENM(self), а потім негайно абстрагує загальне самопосилання. Це має наслідком ідентифікацію самопосилання в двох генераторах і створення нового генератора, параметризованого спільним самопосиланням. З точки зору лямбда-числення, це досить складна операція.

GENPM має два типи клієнтів: підкласи, які можуть успадковувати GENPM шляхом подальшої композиції генератора, та об'єкти, які можуть ВИПРАВЛЯТИ генератор, щоб використовувати його методи. Фіксована точка FIX(GENPM) визначає клас «P, успадкований M». Він пов'язує самопосилання як M, так і P, щоб об'єкт підкласу M міг отримати доступ до методів M і P під час виконання. Відношення b між GENPM та його фіксованою точкою показано

на рисунку 3.14.

Тепер ми можемо виразити різницю між успадкуванням і використанням класу. Коли два класи використовують один одного, ми беремо їхні фіксовані точки перед їх компонуванням, щоб їх композиція була:

compose(FlX(GENP), FIX(GENM))

Композиція не комутує з використанням фіксованих точок, тому ми маємо нерівність: *FIX(COMPOSE(GENP, GENM)) Ф compose(FIX(GENP), FIX(GENM))*

Ліва сторона моделює успадкування P від M, а права сторона моделює P і M, симетрично використовуючи один одного як взаємно рекурсивні класи або функції.

Ця модель успадкування відкриває нові математичні основи для розгляду генераторів як об’єктів першого класу, над якими можна виконувати такі операції, як композиція. Він пов’язує формальне поняття взяття фіксованої точки та неформальне поняття фіксації рекурсивно визначеної ідентичності. Той факт, що фіксовані точки можна просто реалізувати шляхом присвоєння значення змінній покажчика, з’єднує абстрактну математичну теорію з конкретною технікою реалізації.

Аналіз самореференції забезпечує міст до вивчення внутрішньої природи *себе* через рефлексію. Успадкування дозволяє поступове конструювання складних «я», тоді як рефлексія забезпечує обчислювальну структуру для комплексних моделей «я».

3.3 Рефлексія та математичні моделі

Рефлексія людини означає роздуми про власні ідеї, дії та досвід. За аналогією обчислювальне відображення - це здатність обчислювальної системи представляти, контролювати, змінювати та розуміти власну поведінку. Рефлексивні системи полегшують інтроспективні завдання, такі як налагодження, моніторинг, компіляція та виконання. Вони підтримують

еволюцію системи та самореорганізацію. Вони забезпечують додатковий вимір абстракції (мета-абстракцію), яка доповнює -абстракцію даних і суперабстракцію. В об'єктно-орієнтованих системах метаабстракція фіксується метаоб'єктами, абстракція даних – класами, а суперабстракція – успадкуванням. Мета-абстракція за своєю суттю не є самореферентною. Ми можемо представити, контролювати та моделювати систему за допомогою метаопису, відмінного від самої системи. Операційні системи та компілятори, написані мовою асемблера, є мета-абстракціями, які дозволяють нам контролювати та використовувати ресурси комп'ютера для вирішення проблем. У рефлексивній системі метаопис — це а) високорівневий опис у представленні, що використовується для вирішення проблеми, і б) причинно пов'язаний із системою в тому сенсі, що він динамічно контролює, а також статично описує прикладну систему. Рефлексивний метаопис є частиною системи, що описується, і повинен описувати її рефлексивні здібності разом із її здатністю виконувати суттєві завдання. Ідея обчислювальної рефлексії сходить до Lisp, який представляє свої програми у вигляді списків. Інтерпретатор Lisp Apply (див. Рисунок 3.14) може виконувати програми Lisp і, отже, розуміє структуру Lisp. Якщо надати програму та її дані (обидва представлені списками), він застосовує список програм до списку даних, щоб отримати результат. APPLY можна змінити, щоб змінити його власну поведінку виконання, наприклад, вставивши функції моніторингу або налагодження. Універсальні машини Тьюрінга мають відбиваючу структуру, дуже схожу на структуру інтерпретатора Lisp APPLY. Універсальна машина Тьюрінга може взяти представлення машини Тьюрінга та його дані та обчислити ефект застосування машини Тьюрінга до її даних. Він достатньо розуміє представлення машини Тьюринга, щоб виконувати програми, які вони представляють, і може обмірковувати ці програми іншими способами, наприклад, друкуючи сліди обчислень. Тьюрінг продемонстрував певні обмеження на відображення, наприклад неможливість довести, що машина Тьюрінга зупиниться. Результат про неповноту Геделя означає, що відображають математичні системи ніколи не можуть повністю зрозуміти себе.

Компілятори компіляторів - це рефлексивні моделі, які розуміють компіляцію: вони можуть поводитися як конкретний компілятор, якщо їм надається специфікація мови та програма цією мовою. Відбиваючі моделі Ліспа, машин Тьюрінга та компіляторів мають надзвичайно подібну структуру, що відображає те, що відображення має надійну та просту обчислювальну модель.

Рефлексивна система має три компоненти (M, A, D), які ми можемо назвати компонентами мета, агента та даних. Метакомпонент M є метаописом поведінки агентів A, що працюють з даними D. Рефлективні системи також вимагають, щоб M був причинно пов’язаний з A і D у тому сенсі, що він фактично використовується для викликання поведінки. Ключовим завданням проектування є вибір представлення агентів, щоб ними можна було одноманітно та природно маніпулювати метакомпонентом. Ми представляємо програми Lisp списками, програми машини Тьюрінга стрічками, а визначення мови таблицями. Після вибору представлення метакомпонент можна визначити простим способом. Перегляд агентів як даних називається *реифікацією,* оскільки дозволяє розглядати агентів як об’єкти (значення першого класу).

Метакомпонент можна розглядати як *абстракцію другого порядку* (у сенсі функцій другого порядку), оскільки він діє на агентах, які є (уявленнями) абстракцій. Навпаки, функції другого порядку є рефлексивними, оскільки вони можуть впливати на функції та розуміти їх. Ідеї бути рефлексивними, бути другорядними та розглядати агентів як першокласні цінності пов’язані між собою .

Рефлексія - це антропоморфний термін, який описується такими антропоморфними поняттями, як «самопізнання», «розуміння», «поведінка» тощо. Термін «об’єкт» сам по собі є антропоморфним, тому антропоморфні аналогії, запропоновані рефлексією, добре вписуються в об’єктно-орієнтовану парадигму. Антропоморфізм моделює обчислювальні ідеї за допомогою людських аналогій так само, як об’єктно-орієнтоване програмування моделює реальний світ за допомогою обчислювальних аналогій. Ми не приносимо вибачень за антропоморфізм, оскільки моделювання є респектабельною та навіть

центральною науковою технікою. У цьому відношенні ми не згодні з Дейкстрою [Ді].

Системи, що базуються на рефлексивних об’єктах, відображають декілька агентів (класів або об’єктів), а не лише один об’єкт. Метаоб'єкт пов'язаний з кожним класом або об'єктом для представлення його структури та поведінки. Метаоб’єкти містять інформацію про реалізацію та інтерпретацію класів і об’єктів, про те, як вони друкуються, і про те, як створюються нові об’єкти. Протоколи метаоб’єктів можуть надати стислу специфікацію семантики виконання класів і об’єктів. Використання метаоб’єктів для рефлексії в послідовних мовах обговорюється в [Ма] і [Fe].

Метаоб’єкти для об’єктно-орієнтованих паралельних систем повинні представляти не лише методи та стан об’єкта, але також його черги повідомлень і метод оцінки:

змінні екземпляра метаоб'єкта

стан об'єкта, методи об'єкта, черга, оцінювач, методи режиму

моделювати поведінку об’єкта, діючи на змінні екземпляра

Методи метаоб'єкта моделюють надходження, планування та виконання повідомлень. Моделювання черг і повідомлень робить рефлексивне обчислення більш складним, ніж для послідовних систем, але все ще керованим. Після розробки базової архітектури рефлексії можна вказати різні види рефлексивних обчислень, які змінюють поведінку системи. Приклади моніторингу, динамічної модифікації об’єктів і управління часом наведені в [YW],

Об’єктно-орієнтовані рефлексивні системи дозволяють кожному класу мати власну рефлексивну дисципліну для інтерпретації, планування повідомлень, налагодження тощо. Поведінка за замовчуванням може бути визначена в метаметаоб’єкті, який відображає метаоб’єкти та визначає стандартну поведінку, коли нічого не вказано

в метаоб'єкті. Об’єктні системи відображають на двох рівнях: через метаоб’єкти, які відображають окремі класи, і через метаметаоб’єкт, який відображає метаоб’єкти та визначає поведінку за замовчуванням. Також

можливе більш дрібне відображення на рівні окремих об’єктів.

Рефлексивні об’єктно-орієнтовані системи надають модель для слабозв’язаних багатопарадигмальних середовищ, де кожен клас потенційно є парадигмою обчислень, а парадигми взаємодіють шляхом передачі повідомлень. Наприклад, реалізація менеджерів вікон за допомогою класів, протоколів для керування вікнами за допомогою метаоб’єктів і керування вікнами за замовчуванням за допомогою метаметаоб’єктів забезпечує структуру реалізації для мультипарадигмальних кооперативних обчислень для кількох осіб.

Рефлексивна складність є мірою внутрішньої складності системи. Лісп, універсальні машини Тюрінга та лямбда-числення мають винятково низьку відбивну складність. Коли мова стає вищим рівнем, її більша виразність повинна дозволити коротко виразити рефлексивну модель її примітивів високого рівня. Але на практиці примітиви високого рівня рідко мають рефлексивну здатність моделювати себе. Тому мови високого рівня, як правило, мають більшу рефлексивну складність, хоча добре розроблені мови з надійними примітивами мають набагато нижчу рефлексивну складність, ніж погано спроектовані мови.

Не випадково прості рефлексивні системи, пов'язані з машинами Lisp і Turing, нетипові. Типізація руйнує обчислювальну чистоту, вводячи сторонні вимоги сумісності типів, які не можуть бути виражені в обчислювальному формалізмі нетипізованої системи. Відображення обох типів і обчислень вимагає, щоб рефлексивна система обробляла два дуже різні обчислювальні механізми та взаємодію між ними. Безпека часу виконання типізованих систем реалізується за рахунок рефлексивної простоти.

Відбиваючі системи полегшують як практичні обчислення, так і концептуальне розуміння систем, на яких вони відбиваються. Вони сприяють розумінню системи як комп’ютерами, так і людьми. Метакомпоненти визначають операційну семантику агентів. Оскільки складність специфікації операційної семантики пов’язана з властивою концептуальною складністю людського розуміння, рефлексивна складність забезпечує міру концептуальної складності. Набагато більша рефлексивна складність типізованих систем

порівняно з нетипізованими системами дає міру величезних концептуальних витрат на впровадження типів.

3.4 Висновки по розділу

Виведення типів у мовах програмування забезпечує баланс між безпекою типізації та гнучкістю без необхідності явних декларацій. Математичні моделі, зокрема фіксовані точки та генератори, пропонують точну основу для аналізу рекурсії, успадкування й поведінки самопосилань, що дозволяє точніше інтерпретувати як типізовані, так і нетипізовані конструкції мов.

ВИСНОВКИ

Вибираючи серед наведених вище сценаріїв, ми нескромно припускаємо, що об’єктна орієнтація насправді може бути постійним і навіть фундаментальним модним словом. Опис області дискурсу поведінковими атрибутами її сутностей є природним для будь-якої області застосування. Різниця між інкапсульованою внутрішньою поведінкою та зовнішнім зв’язком через інтерфейси відображає те, що в будь-якій організації (банку) чи організмі (людському тілі) між переплетеними внутрішніми субкомпонентами та явними інтерфейсами із зовнішнім середовищем. Різниця між декларативним описом поведінки класами та імперативним створенням екземплярів для конкретних обчислень також є природною. Спадкування – це дещо більш спеціалізоване поняття, яке полегшує повторне використання, композицію та модифікацію поведінки, визначеної класами. Представлення класів об’єктами дозволяє керувати поведінкою та маніпулювати нею так, ніби це дані, забезпечуючи - альтернативу успадкуванню для радикальних обчислень у представленнях поведінки. Об'єктно-орієнтована парадигма підтримує кілька, але скоординованих парадигм мислення та вирішення проблем. Він чудово врівноважує парадигми переходу між станами, зв’язку та класифікації ¬, поєднуючи обов’язкові обчислення всередині об’єктів і передачу повідомлень між об’єктами з декларативною специфікацією поведінки об’єктів. Він «закритий» для рефлексії в тому сенсі, що метакласи та метаметакласи самі по собі є об’єктами, і полегшує незалежні протоколи рефлексивної інтерпретації для різних об’єктів і класів. Це сприяє гармонійній інтеграції абстракції даних, супер-абстракції та мета-абстракції в комплексну обчислювальну структуру. Його універсальність як надійної техніки представлення, моделювання та абстрагування свідчить про те, що об’єктно-орієнтована парадигма є концептуально та обчислювально фундаментальною. Об'єктно-орієнтовані системи підтримують взаємодію слабко пов'язаних розподілених агентів, що виконуються одночасно. Індивідуальні агенти можуть моделюватися

автоматами, але математика асинхронно взаємодіючих агентів не настільки добре розроблена, як для атомарних послідовно виконуваних агентів. Якщо агентура має бути замкнута на композицію, так що колекції асинхронно співпрацюючих агентів моделюються як агенти, поняття агента та еквівалентності між агентами стає більш складним. Зверніть увагу, що моделювання суспільств агентів як агентів за своєю суттю є складним і є предметом дослідження в таких дисциплінах, як соціологія, економіка та когнітивна наука. Об’єктно-орієнтовані системи, як правило, мають глобальні типи та класи: класи об’єктно-орієнтованої бібліотеки зазвичай не розрізняють різні типи клієнтів. Програмування в дуже великому масштабі (мегапрограмування) може порушити це припущення, оскільки воно стосується систем, розроблених різними організаціями, що мають різні концептуальні рамки та звертаються до різних областей застосування. Об’єктно-орієнтовані системи забезпечують відправну точку для моделювання взаємодії неоднорідних компонентів, але їх необхідно розширити, щоб підтримувати неоднорідність представлень даних і систем типів. Об'єктно-орієнтоване програмування є більш конкретним і всеосяжним у своєму рецепті вирішення проблем, ніж структурне програмування. Структурне програмування стосується «структури» загалом, тоді як об’єктно-орієнтоване програмування зосереджується на конкретній формі структури: тій, що пов’язана з об’єктами. Настав час кинути виклик тим, хто стверджує, що «всі говорять про об’єктно-орієнтоване програмування, але ніхто не знає, що це таке». Це вже не відповідає дійсності, оскільки зростає кількість літератури, включно з цією статтею та багатьма цитованими посиланнями, яка визначає, що це таке та як це можна використовувати для вирішення проблем. Об’єктно-орієнтоване програмування — це не тимчасове модне слово, а фундаментальна структура моделювання, яка має щось фундаментальне сказати про обчислення та побудову моделей, і водночас лежить на критичному шляху до підвищення продуктивності програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Abelson, H., & Sussman, G. J. *Structure and Interpretation of Computer Programs.* — 2nd ed. — Cambridge, MA: MIT Press, 1996. — 657 с. — Режим доступу: https://mitpress.mit.edu/9780262510875

2. Bird, R. *Introduction to Functional Programming Using Haskell.* — 2nd ed. — London, UK: Prentice Hall, 1998. — 464 с.

3. Bloch, J. *Effective Java.* — 3rd ed. — Addison-Wesley, 2018. — 412 с.

4. Chiusano, P., & Bjarnason, R. *Functional Programming in Scala.* — Shelter Island, NY: Manning Publications, 2014. — 320 с.

5. Fogus, M., & Houser, C. *Functional JavaScript: Introducing Functional Programming with Underscore.js.* — Sebastopol, CA: O'Reilly Media, 2013. -260 с.

6. Fowler, M. *Refactoring: Improving the Design of Existing Code.* — 2nd ed. — Addison-Wesley, 2018. — 448 с.

7. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* — Addison-Wesley, 1994. — 416 с.

8. Hudak, P. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 1989, 21(3), 359–411. — Режим доступу: https://doi.org/10.1145/72551.72554

9. Hughes, J. Why functional programming matters. *The Computer Journal*, 1989, 32(2), 98–107. - Режим доступу: https://doi.org/10.1093/comjnl/32.2.98

10. Jensen, K. Functional vs. object-oriented programming: A comparative study. *Journal of Software Engineering Research*, 2023, 10(2), 45–60. — Режим доступу: https://doi.org/10.1016/j.jser.2023.02.003

11. Kay, A. C. The early history of Smalltalk. *ACM SIGPLAN Notices*, 1993, 28(3), 69–95. — Режим доступу: https://doi.org/10.1145/155360.155364

12. Lipovača, M. *Learn You a Haskell for Great Good!: A Beginner's Guide.* — San Francisco, CA: No Starch Press, 2011. — 400 с.

13. MacLennan, B. J. *Functional Programming: Practice and Theory.* - Addison-Wesley, 1990. — 608 с.

14. Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship.* — Prentice Hall, 2008. — 464 с.

15. Meyer, B. *Object-Oriented Software Construction.* - 2nd ed. - Prentice Hall, 1997. — 1296 с.

16. Microsoft. F# Documentation: Functional Programming Concepts. *learn.microsoft.com*, 2024. — Режим доступу: https://learn.microsoft.com/en-us/dotnet/fsharp

17. O'Sullivan, B., Goerzen, J., & Stewart, D. *Real World Haskell.* — Sebastopol, CA: O'Reilly Media, 2008. — 720 с. — Режим доступу: http://book.realworldhaskell.org

18. Petricek, T. *Real-World Functional Programming: With Examples in F# and C#.* — Shelter Island, NY: Manning Publications, 2009. — 560 с.

19. Petricek, T., & Skeet, J. Comparing functional and object-oriented paradigms in .NET. *Proceedings of the 2021 .NET Conference*, 2021, 88–95. — Режим доступу: https://doi.org/10.1109/NETCONF.2021.9345678

20. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. *Object-Oriented Modeling and Design.* — Prentice Hall, 1991. — 512 с.

21. Sestoft, P. Functional programming vs. object-oriented programming: A performance comparison. *Nordic Journal of Computing*, 2019, 15(4), 123–140. - Режим доступу: https://doi.org/10.1007/s10664-019-09724-5

22. Stroustrup, B. *The C++ Programming Language.* - 4th ed. -Addison-Wesley, 2013. - 1376 с.

23. Thompson, S. *Haskell: The Craft of Functional Programming.* - 3rd ed. - Boston, MA: Addison-Wesley, 2011. — 528 с.

24. Wadler, P. Monads for functional programming. *Advanced Functional Programming*, 1995, 24–52. - Режим доступу: https://doi.org/10.1007/3-540-59451-5_2

25. Wlaschin, S. *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#.* - Raleigh, NC: The Pragmatic Bookshelf, 2018. — 312 с.

26. X Post by @prog_philosophy. Functional vs OOP: Why not both? Hybrid approaches in modern software design. *X*, 2024. - Режим доступу: https://t.co/ZxYpQwRtYp

27. Yorgey, B. A. Comparing paradigms: Functional and object-oriented programming in practice. *Journal of Functional Programming*, 2022, 32, e15. - Режим доступу: https://doi.org/10.1017/S0956796822000090