



synrc research center s.r.o.
RONÁČOVA 141/18, PRAHA 3 13000, CZECH REPUBLIC

**Системна інженерія та верифікація
уніфікованого обчислювального середовища**

**System Engineering and Verification
of Unikernel Execution Environment**

Павло Маслянко, Київський Політехнічний Інститут

Максим Сохацький, Synrc Research Center

Листопад 2015

Содержание

1	Вступ	3
1.1	Системна інженерія	3
1.2	Уніфіковані обчислювальні середовища	3
1.3	Верифікація програмного забезпечення	4
1.4	Компактні адаптивні протоколи	4
1.5	Завдання дослідження	5
1.5.1	Зберігання обчислень	6
1.5.2	Виконання процесів	6
1.5.3	Комутація протоколів	6
1.5.4	Розподілені у просторі та часі системи	6
1.5.5	Різні аксіоматичні рівні	6
1.6	Структура роботи	7
1.6.1	Уніфіковані обчислювальні середовища	7
1.6.2	Формальні методи	8
1.6.3	Робоча теорія	8
1.6.4	Формальна підсистема	8
1.6.5	Результуюча мова	8
1.7	Додаткові та дотичні теорії	9
1.7.1	Теорія масового обслуговування	9
1.7.2	Мережі Петрі	9
1.7.3	Теорія категорій	9
1.7.4	Темпоральна логіка	9
1.8	Основні теорії	10
1.8.1	π числення	10
1.8.2	λ числення	10
1.8.3	Інтуїціоністична теорія типів Мартіна Льюфа	10
1.9	Предмет дослідження	11
1.9.1	Алгебраїчний підхід	12
1.9.2	Напівгрупа активностей	12
1.9.3	Лінійність обчислень	13
1.9.4	Управління ефектами	13
1.9.5	Визначення процесу	15
1.9.6	Управління процесами	15
1.9.7	Алгебра процесів	15
1.9.8	Лямбда числення	16
1.9.9	Алгебраїчні типи даних	16
1.9.10	Процеси і протоколи	16
1.9.11	Логіка та квантори	17
1.9.12	Контраваріантні процеси	17
1.9.13	Типи процесів	17
1.10	Віртуальна машина	18
1.11	Сигнатури сервісів	18
1.12	Мета дослідження	19
1.13	Результати	19

1 Вступ

1.1 Системна інженерія

Протягом історії обчислювальної техніки було створено різні класи та способи обчислень, різні теорії та підходи до програмування таких систем, різні класи систем програмування. Зараз уже стало зрозумілим, що інженіринг систем які не піддаються до верифікації формальними методами не може бути застосований у галузях де вимоги до якості особливо підвищені, як то космонавтика, енергетика та телекомунікації.

1.2 Уніфіковані обчислювальні середовища

Називатимемо програмні комплекси та системи, які розповсюджуються на всі прошарки моделі OSI вище фізичного, уніфікованими середовищами. Перелічуючи проекти, які вплинули на хід історії обчислювальної техніки, серед такого уніфікованих замкнених систем, можна згадати Smalltalk-80, створений в Xerox Parc Аланом Кеєм, якого можна назвати творцем об'єктно-орієнтованого підходу, за що він був удостоєний премії Тюрінга. Віртуальні машини середовища Smalltalk представляють відносно ізольовано по відношенню до операційної системи систему типів разом з моделлю акторів, що дозволяє самостійно розподіляти час обчислювального середовища. Стан таких віртуальних апаратно незалежний та може бути перенесений на віртуальні машини які працюють на процесорах інших архітектур. Інший клас систем які були представлені у той час – це Lisp машини. Вони теж були здатні зберігати стан своєї віртуальної машини яка, як і усе середовище була написана на мові Lisp. Серед експериментальних та науково-дослідних уніфікованих систем можна відзначити Singularity від Microsoft Research, яка давно є постачальником високоякісних теоретико-практичних інструментів по верифікації програмного забезпечення. Сучасні уніфіковані системи існують на трьох мовах програмування (на OCaml це MirageOS, на Haskell це HaLVM, для Erlang це віртуальна машина LING, яка створена в Україні) та здатні виконуватися без операційної системи. В даній роботі дається обґрунтуванню вибору віртуальної машини Erlang та її операційної семантики у якості основи для обчислювального середовища та його сервісів.

Застосовуючи формальні методи доведення коректності велику частину має повнота за макненість системи, адже недоведені або неверифіковані частини можуть вплинути на детермінованість а отже якість системи. Тому має велике значення забезпечення виконання системи якомога ближче до апаратного забезпечення. Основні напрямки побудови замкнених середовищ на функціональних мовах програмування існують для мов Erlang, Haskell та OCaml.

1.3 Верифікація програмного забезпечення

За багато років кількість теорій, які використовуються для побудови програмного забезпечення значно розширилися: починаючи з теорії компіляції сучасних функціональних мов, та систем програмування на основі теорії типів, включаючи сучасні моделі обчислень, які побудовані на основі лямбда числення та числення процесів, закінчуючи віртуальними машинами які працюють у семантиці захищених, простих за структурою процесів, час яких розподіляється у прозорий та ефективний спосіб.

Розглядаючи системи, які піддаються формальній верифікації, кажуть про сертифіковане або верифіковане програмне забезпечення. Серед логічних систем, які застосовувалися для цього можна відзначити клас темпоральних логік для доведення цілісності розподілених у просторі та часі систем (приклад таких систем: NuPRL від університету Корнела в Нью-Йорку; TLA+ фреймворк Леслі Лампорта за який він отримав премію Тюрінга від Microsoft Research), а також системи з залежними типами (теорія типів Мартіна Льофа) побудовані на основі числення індуктивних конструкцій (приклад таких систем: Coq побудована на мові OCaml від національного науково-дослідного інституту Франції INRIA; Agda побудовані на мові Haskell від шведського інституту технологій Чалмерс; Lean побудована на мові C++ від Microsoft Research та Університету Каргені-Мелона; Idris побудована на мові Haskell Едвіна Бреді з шотландського Університету ім. св. Андрія).

1.4 Компактні адаптивні протоколи

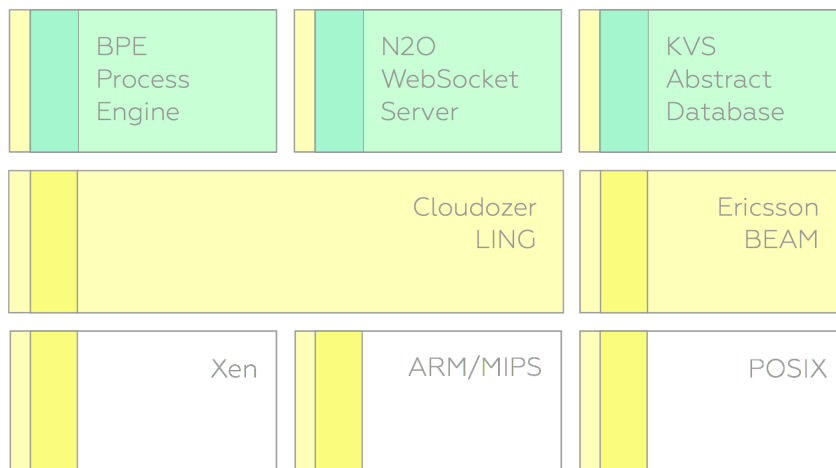
Зараз на планеті налічується близько 50 мільярдів пристроїв які повинні обслуговуватися масштабованими ефективними серверними комплексами які гарантовано не виконують додаткових та непотрібних обчислень. Це потребує повного перепроектування усіх протоколів які виникають між компонентними та підсистемами. Ця робота також представляє уніфікований та простий стек протоколів розроблюваного обчислювального середовища.

heart	ping, n2o, io
spa	client, server
bin	bin, ftp
nitrogen	pickle, direct, ev
bpe	run, complete, amend, until, start
mq	pub, sub, unsub, msg, resume, suspend
roster	friend, confirm, private, typing, presence
muc	join, room, public, leave
rest	get, post, exists, put, options, delete
search	mark, query

1.5 Завдання дослідження

Тактична мета даної роботи – це реалізація усіх верхніх компонентів OSI на базі формальної теорії та мови з залежними типами. Побудова зручної сучасної гнучкої верифікованої теорії з компактною системою типів є основним завданням даної роботи. Будемо орієнтуватися на мінімально можливу систему типів та набір функцій які забезпечать успішне промислове впровадження кінцевого продукту з компактною, проте достатньо потужною системою типів.

В першу чергу мова йде про три найголовніші сервіси: збереження інформації, система управління процесами та мультиплікатор протоколів, який у даній системі відіграє роль сервера додатків. Успішна реалізація цих трьох систем дасть нам змогу будувати більш складні та масштабовані транзакційні системи та сервіси на базі створеної формальної теорії, а також прокладе шлях подальшому розвитку даного теоретико-практичного комплексу.



Апаратура Що стосується апаратної частини моделі OSI, то тут ми будемо вважати умовно, що апаратне забезпечення і так проходить усі належні види моделювань, такі як розрахункові імітаційні та інші. Крім того як ми знаємо верифіковані моделі уже застосовуються для розробки мікропроцесорів.

Віртуальна машина Верифікована система типів середнього рівня моделі OSI — віртуальної машини є предметом майбутніх досліджень. Адам Чіпала [25] показав як можна верифікувати виконання команд віртуальної машини та компіляцію цих команд в байт-код.

Сервіси Верифікована система типів верхнього рівня моделі OSI — це центральний фокус даної роботи. Ми розглянемо декілька основних сервісів які забезпечують персистентність, виконання процесів, та мультиплікацію протоколів.

1.5.1 Зберігання обчислень

У цій роботі ми будемо верифікувати системні бібліотеки для майбутньої верифікованої віртуальної машини. Зокрема значна увага буде приділятися послідовностям. Система персистентного зберігання послідовностей є ядром нашого обчислювального середовища. Властивості цієї підсистеми є основоположними для перевірки консистентності операційних логів розподілених баз даних.

1.5.2 Виконання процесів

Тут ми дамо формальне визначення та верифікуємо типи системи управління структурованими процесами або FSM автоматами. Бізнес-процеси це зручний та ефективний спосіб упорядкування та формалізації потоків даних. Ми зосередимося на найкомпактніших алгебрах.

1.5.3 Комутація протоколів

Ця частина дасть відповідь на питання структурування протоколів та визначення універсального мультиплікатора. Буде описаний повний стек протоколів для IoT та WebSocket додатків.

1.5.4 Розподілені у просторі та часі системи

Сучасний розвиток техніки та теоретична межа швидкості обробки процесорів вивів на передній план алгоритми та структури даних які ефективно використовують розподілені у просторі та часі ресурси, як то об'єми пам'яті та обчислювальні потужності. Принципи та підходи паралельного та узгодженого програмування дають змогу масштабувати системи та обчислення, однак анонсують нові теорії для забезпечення коректності в умовах підвищеної складності алгоритмів у розподілених системах, такі як алгоритми забезпечення консистентності та транзакційності у розподілених системах PAXOS та CR. Сучасні обчислювальні середовища повинні ефективно управляти великими масивами даних та необмежено масштабуватися. Розподілена архітектура сервера транзакцій які працюють поверх формального верифікованого сховища даних буде представлена окремою роботою. Даний прототип існує у вигляді координатора транзакцій який працює за алгоритмом ланцюгової реплікації і використовує семантику сховища, яке у формальному вигляді представлене у цій роботі.

1.5.5 Різні аксіоматичні рівні

Важливо додати, що з формальної точки зору потрібно щоби на границях трьох системних рівнів формальні типи дотичних логічних теорій співпадали. Для доведення коректності формалізованої віртуальної машини достатньо визначити систему типів команд мови процесор та типи їх операндів. А для доведення коректності верхнього рівня достатньо аксіоматизувати дотичну формальну теорію віртуальної машини. Тому в даній роботі буде представлена така публічна система типів віртуальної машини що одразу відкриє дорогу для деталізації і подальшої теоретичної розробки формальної теорії для віртуальної машини.

1.6 Структура роботи

Покладаючись на аналіз рішень в області уніфікованих середовищ та набір математичного і лінгвістичного забезпечення у якості формальних методів, ми будемо формувати формальну теорію та бібліотеку типів для формальної реалізації ключових підсистем.

Усі існуючі уніфіковані системи, оскільки є замкненими, пропонують повністю свій інтерфейс взаємодії, свою операційну семантику та свою систему типів. У цьому сенсі наша система не є винятком і пропонує повний стек програмного забезпечення, ключові частини якого ми намагатимемося формально описати.

На основі створеної базової теорії та її бібліотеки типів ми створимо формальними методами доказову базу для кожного компонента обчислювального середовища, який включений в дану роботу: **зберігання обчислень, виконання процесів, мультиплікатор протоколів.**

Далі кожна така верифікована система буде транслюватися в цільову мову віртуальної машини обчислювального середовища, яка у загальному випадку може відрізнятися від вибраної нами Erlang.

Особливість цієї уніфікованої системи в повному та тотальному перегляді усіх існуючих рішень включно з базовими протоколами інтернету типу HTTP. Так наприклад для сервера додатків у якості транспортного протоколу використовується бінарний WebSocket протокол. Усі протоколи формуються у вигляді алгебр над станами процесів.

1.6.1 Уніфіковані обчислювальні середовища

Об'єктом дослідження є усі можливі моделі обчислювальних середовищ в основному придатних до верифікованого аналізу та обробки формальними методами. Ми вибрали середовище основані на віртуальних машинах зі своєю системою типів та байткодом, які уже використовуються у промисловій експлуатації в телекомунікаційній сфері та є лідерами у цій області це телекомунаційна платформа Ericsson – Erlang/OTP. Ступінь проникнення цієї технології у сучасних телекомунікаціях достатньо високий, особливо завдяки розвиненій підтримці ASN.1, SNMP, H.262, Radius та інших промислових стандартів у телекомунікаційній сфері. Перші успішні масштабні проекти на Erlang були створені ще у 1986 році, а зараз Erlang демонструє успішність завдяки таким проектам як WhatsApp, який обслуговує 30 мільярдів повідомлень в день у порівнянні з SMS трафіком, який генерується щоденно у розмірі 20 мільярдів повідомлень.

1.6.2 Формальні методи

Самі формальні методи теж є частиною об'єкта дослідження. Ми досліджуємо ті структури та алгоритми які дадуть максимально ефективний спосіб кодування та виконання, забезпечуючи при цьому семантику, яка використовується для машинного доведення коректності роботи алгоритмів та узгодженості міжпроцесних протоколів. Ми будемо розробляти робочу формальну теорію за допомогою індуктивних типів, переводячи результати на мову обчислювального середовища, Erlang.

Побудова розподілених та паралельних, тобто здатних виконуватися на багатьох машинах одночасно, та узгоджених, тобто не блокуючих, а значить лінеаризованих систем управління процесами є побічним результатом який очікується від цієї роботи. Усі розподілені у просторі та часі протоколи ми будемо верифікувати використовуючи темпоральну логіку за допомогою системи TLA+.

1.6.3 Робоча теорія

Вибравши конкретну систему автоматичного доведення теорем на основі теорії індуктивних типів, ми будемо основні типи нашої теорії і визначаємо у нашій теорії сигнатури сервісів нашого обчислювального середовища. Досліджуємо побудовану систему типів на предмет відповідності до робочого прототипу та доводимо певні базові властивості які є спільними для кожного із трьох сервісів включених до даної роботи. Виберемо для опрацювання робочої теорії систему автоматичного доведення теорем Lean від Microsoft Research в якості мови з залежними індуктивними типами.

1.6.4 Формальна підсистема

Кожну окрему підсистему ми будемо відокремлювати та визначати які теореми та які констукції використовуються для побудови підсистеми формальним способом, використовуючи математичні конструкції робочої теорії. У нашому випадку є три типи моноїдальних систем на основі яких ми можемо конструювати більш складні кінцеві системи-продукти для кінцевого користування та експлуатації. Для кожної підсистеми з цих трьох застосовується подібна методологія доведення коректності та трансляції математичної моделі в результуючу мову віртуальної машини.

1.6.5 Результуюча мова

Розроблені теореми та сигнатури формальної підсистеми потім компілюються або транслуються в результуючу мову віртуальної машини обчислювального середовища. В нашому випадку це мова Erlang та віртуальні машини BEAM від Ericsson, та LING від Cloudozer. Для цього нам в базовій теорії доведеться формально описати базові примітиви віртуальної машини Erlang та взяти їх у якості аксіом для нашої теорії.

1.7 Додаткові та дотичні теорії

У таксономії теоритичних сутностей умовно визначатимемо графічні топологічно-структурні, аналітичні, статистичні та логічні типи теорій, які ми використовуватимемо для опису комплексної теорії верифікованих середовищ послідовних обчислень розподілених узгоджених паралельних систем.

1.7.1 Теорія масового обслуговування

Теорія масового обслуговування застосовується для побудови статистичних моделей та запобігання відмов. Перші роботи у цій області належать шведському математику Агнеру Крурупу Ерлангу, який займався дослідженнями трафіка у телефонних мережах. Модель масового обслуговування достатньо адекватно описує роботу віртуальної машини Erlang, де клієнти – це процеси, які мають черги повідомлень, та здатні відправляти заявки на обслуговування у такі самі черги інших процесів. Ці заявки, чи повідомлення складають певний протокол взаємодії у системі таких процесів. Тому тут теорія масового обслуговування застосовується для визначення пропускну здатності системи.

1.7.2 Мережі Петрі

Мережі Петрі в даній роботі використовуються як прототип графічного лінгвістичного засобу структури категорій та системи типів. Оскільки такі графічні засоби як UML та різноманітні окремі технологічні стандарти як то BPMN більше допомагають ніж мішають, була розроблена також і графічна мова на базі графічної мови мереж Петрі, оскільки їх семантика ділить один простір з тематикою даної роботи. Ми візьмемо лінгвістичне забезпечення Мереж Петрі як прототип для нашої власної мови візуалізації структур обчислень.

1.7.3 Теорія категорій

Теорія категорій як робоча структурна теорія системи типів мови Eхе та категоріальна семантика числення процесів. Можна було би використовувати абстрактну алгебру загалом, та теорія напівгруп, а саме напівгруп активностей, проте ми будемо використовувати категоріальну семантику, що стало можливо завдяки роботам Лавіра та іншим.

1.7.4 Темпоральна логіка

Темпоральна логіка як індуктивна теорія верифікації розподілених алгоритмів застосовується до доведення коректності усіх нормалізованих підсистем. На основі теорії Леслі Лампорта [7], за яку він отримав премію Тюрінга. Елементи темпоральної логіки нам знадобляться для розробки теорій розподілених у просторі та часі алгоритмів.

1.8 Основні теорії

1.8.1 π числення

Теорія Пі-числення Роберта Мілнера є основним формалізмом обчислювальної теорії та її імплементації. З часів виникнення CSP числення розробленого Хоаром, Мілнеру вдалося значно розширити та адаптувати теорію до сучасних телекомунікаційних вимог, як наприклад хендовери в мобільних мережах. Основні теорми в моделі Пі-числення стосуються непротиворечивості та неблокованості у синхронному виконанні мобільних процесів. Так як сучасний Web можна розглядати як телекомунікаційну систему, тому у розробці додатків можна покладатися у тому числі і на такі моделі як Пі-числення.

1.8.2 λ числення

Лямбда-числення як основна абстракція обчислювальної віртуальної машини. Будучи внутрішньою мовою декартово-замкненої категорії лямбда числення окрім змінних та констант у вигляді термів пропонує операції абстракції та аплікації, що визначає достатньо лаконічну та потужну структуру обчислень з функціями вищих порядків, та метатипизаціями, такими як System F, яка була запропонована вперше Робіном Мілнером в мові ML, та зараз присутня в більш складних, таких як System F ω системах Haskell та Scala. Наша результуюча система типів буде тежована система типів мови Erlang.

1.8.3 Інтуїціоністична теорія типів Мартіна Льюфа

Системи з залежними типами як верифікаційні математичні формальні моделі для доведення коректності. Система Σ та Π типів, як кванторів існування та узагальнення. Системи Mizar, Coq, Agda, Idris, F*, Lean. Ми будемо використовувати автоматизовану систему теорем Lean від Microsoft Research.

Розбудовуючи певний фреймворк чи систему конструктивними методами так чи інакше доведеться зробити певний вибір у мові та способі кодування. Так при розробці теорії абстрактної алгебри в Coq були використані поліморфні індуктивні структури [26]. Однак Agda та Idris використовують для побудови алгебраїчної теорії типи класів, а у Idris взагалі відсутні поліморфічні індуктивні структури та коіндуктивні структури. В Lean теж відсутні коіндуктивні структури проте повністю реалізована теорія HoTT на нерекурсивних поліморфних структурах що об'єднує основні чотири класи математичних теорій: логіка, топологія, теорія множин, теорія типів. Як було показано Стефаном Касом [28], одна з стратегій імплементації типів класів — це використання поліморфних структур. Хоча в Lean також підтримуються типи класів нами була вибрана стратегія імплементації нашої теорії з використанням нерекурсивних індуктивних структур, що дозволить нам оперувати з персистентними структурами на низькому рівні. Крім того такий спосіб кодування ієрархій повністю відповідає семантиці Erlang, де немає типів класів, а дані передаються запаковані в кортежі-структури.

1.9 Предмет дослідження

Предметом дослідження даної роботи є розробка формальних методів для побудови операційних середовищ та метамodelей для їх формальної специфікації. Розглянувши усі можливі математичні моделі опису формалізму процесів ми формуємо ряд вимог корисних для побудови ефективної моделі яка дозволить:

- Скоротити об'єм коду на порядок
- Нормалізувати дані для їх статистичної обробки
- Легко обчислювати показники системи масового обслуговування
- Легко доводити коректність
- Мінімізувати цикл розробки програмного забезпечення
- Підвищити ефективність виконання

Для забезпечення стратегії цієї роботи ми визначаємо найбільш оптимальну модель потоку даних та функції для їх обробки, використовуючи алгебраїчний підхід для опису протоколів та їх категорій. Серед основних фаз дослідження можна виділити:

- Дослідження алгебраїчних структур середовища
- Визначення характеристик нормалізованих та оптимальних структур
- Побудова системи типів
- Визначення властивостей системи та доведення теорем
- Розробка системи додатків на основі метамodelі на мові Erlang
- Впровадження, діагностика та аналіз роботи системи

Іншими словами словом ми розроблюємо теорію та імплементацію мінімалістичного сертифікованого верифікованого операційного середовища для компонентів замкненої системи: абстракції апаратного забезпечення, мова програмування, віртуальна машина, комунікаційні черги, бази даних, веб сервери, сервери додатків, та усе, що піддається верифікації та по можливості є коректним за побудовою. Починаючи з фундаментального формалізму лямбда та пі числення, через абстракції віртуальної машини до віртуальної апаратури, реальні сучасні додатки та протоколи, закінчуючи прикладами, засобами та документацією на створене обчислювальне середовище та його модель.

1.9.1 Алгебраїчний підхід

Ми будемо застосовувати алгебраїчні рекурсивні типи даних для побудови системи типів, тому зручно буде також застосовувати алгебраїчний підхід до генералізації теорії процесів. З алгебраїчної точки зору процеси, або кінечні автомати, представляють собою напівгрупи дій. З категоріальної точки зору процеси — це функтори які перетворюють категорії з декартовими добутками типів протоколу та стану процесу на категорію станів. Кожен процес визначається таким функтором, адже простір дій Σ для кожного процесу є унікальний.

$$\begin{array}{lll} A & : & \Sigma \times X \rightarrow X \\ X & : & \Sigma \times \Lambda_X \\ & | & \perp \\ \Sigma & : & \begin{array}{ll} ok & \times _ \\ error & \times _ \\ io & \times _ \times _ \end{array} \end{array}$$

У функціональних мовах програмування така категорія будемо мати вигляд програми, де функтор буде представлений функцією перетворення станів процесу, а протокол буде типом-сумою усіх можливих запитів до процесу:

```

action : proto → state → state
state  : list proto
definition sigma := ok + error + io

record proto := (data: iterator)
record ok    extends proto
record error extends proto
record io    extends proto := (effects: iterator)

```

1.9.2 Напівгрупа активностей

Дана алгебраїчна модель визначає достатньо просту та визначену структуру. Мета нашої структури забезпечити детермінованість послідовності, її початковий елемент та термінальний.

$$\begin{array}{ll} spawn & : \top \rightarrow S \\ run & : S \rightarrow \perp \\ \perp & : \odot S \rightarrow S \\ complete & : \otimes S \rightarrow S \end{array}$$

Одиниця напівгрупи демонструється номінально як операція запиту поточного стану процесу. Інша бінарна операція напівгрупи по відношенню до протокола та стану процесу *complete* передбачає виконання функції активності поточного кроку процесу, яка визначена в сигнатурі як $action : \Sigma \rightarrow P \rightarrow P$.

```

record app (P S: Type) := (spawn : P → S)
                        (run    : S → P)
                        (action  : P → S → S)

```

1.9.3 Лінійність обчислень

Списки є фундаментальними структурами та найпростішим рекурсивним алгебраїчним типом. Природа обчислень, як результат виконання процесів теж лінійна. Можна навіть визначити оператор похідної, як функтор, який для алгебраїчного типу процесу буде давати список, а для списку буде давати скаляр. Як було показано [14] мережі петрі, а значить і процеси є моноїдами які генерують лінійну послідовність подій.

$$\begin{aligned}\Delta \Lambda^0 &: \times^0 \\ \Delta \Lambda^n &: \Lambda^{n-1}\end{aligned}$$

Саме лінійність кінцевого процесу виконання певного структурованого або циклічного алгоритму відіграє ключову роль у моделюванні віртуальних обчислювальних середовищ. Такі лінеаризовані системи уже показали свою ефективність в певному класі обчислень, таких як ланцюгова реплікація, реактивні системи, та інші моделі напівгруп навколо певних типів – протоколів взаємодії між процесами. Крім того такі послідовності подій піддаються статистичній обробці для визначення первних кластеризацій та інших кореляцій у просторі та часі, тому можна говорити про певні зручні, нормалізовані системи типів для такого роду маніпуляцій над даними.

1.9.4 Управління ефектами

Однак що робити коли обчислення в моноїдальних категоріях проходять з помилками? Як обробляти ці помилки, та як ці помилки пов'язані зі слідом процесу? Таким чином кажуть про ефекти які можуть виникнути при роботі чистих функцій. Кодування ефектів за допомогою типів зараз є стандартом де-факто в сучасних функціональних мовах програмування. Так наприклад в мові Haskell ефекти кодуються за допомогою монадичних типів. Однак, насправді, достатньо було б використовувати менш загальний спосіб кодування ефектів у вигляді списків, з огляду на лінійність процесу обчислень. Так в мові Idris ефекти середовища дійсно представлені у вигляді списків. Як показано [29], ефекти навіть вносяться як мовні елементи у функціональну мову, як це наприклад існує в імперативних мовах у вигляді типів виняткових ситуацій.

Класично до ефектів, крім помилок та виключень, відносять також усі операції що взаємодіють зі змінними за межами локального контексту виконання функції (параметри чи локальні змінні в стеку функції). Також до ефектів відносять стан процесу що знаходиться контекстом вище ніж контекст даної функції. Однак в нашій моделі стан включений явно у сигнатури наших моноїдальних конструкцій.

Ми хочемо зробити так, аби ефекти були явними готовими до персистентності. Наприклад щоби можна було взяти послідовність ефектів у вигляді списку і передати їх по мережі чи зберегти у сховищі середовища. Таким чином ми пропонуємо відкриту модель кодування ефектів у вигляді дво-зв'язних списків (сигнатура бінарного дерева з аксесорами *next* та *prev*).

Мотивація такого підходу полягає в тому, що для транзакційних систем в яких існує лог операцій, цей лог операції і є ефектами які виникають в процесі обчислень. І в розподілених системах зручно явно оперувати такими ланцюгами ефектів. Тому ми описуємо універсальний механізм зберігання поліморфних структур на базі абстрактного key-value яке нам пропонує віртуальна машина, і ефекти ми будемо кодувати індуктивними поліморфними структурами. Кожна послідовність ефектів та обчислень кодується певним типом та зберігається у сховищі під унікальним ключем. В процесі роботи система оепрує обчисленнями як послідовностями, і може їх склеювати, підрізати, пробігати та згортати.

Ми почнемо з двох найголовніших послідовностей *ok* (послідовність обчислень) та *error* (послідовність помилок), а також об'єднана стурктура *io* яка тримає обидва вида ефектів. Результат обчислень кожної функції (*ok* чи *error* в залежності від результату) дописується у відповідний ланцюг контейнера ефектів який знаходиться близько до процесу який тримає стан і виконує дану функцію.

<i>KVS</i>	:	<i>get</i>	×	—
	:	<i>put</i>	×	—
		<i>add</i>	×	—
		<i>remove</i>	×	—
		<i>cut</i>	×	—
		<i>append</i>	×	—
		<i>fold</i>	×	—

```

definition ids          := option ℕ
  record container := (top: ids) (size: ℕ)
  record iterator  := (id: ids) (next: ids) (prev: ids)

  record db          extends proto
  record get          extends db
  record put          extends db
  record add          extends db
  record cut          extends db
  record remove       extends db
  record append       extends db
  record fold         extends db
definition kvs := get + put + add + remove +
  append + fold + ok + error + io

  record store
extends app kvs container :=
  (sup: list container)
  (tab: list table)
  (calculation: iterator → container → container)

```

1.9.5 Визначення процесу

Щоб формально визначити процес ми спочатку визначимо домени або типи, які фігурують при визначенні процесу.

Errors	: #error
Sucesses	: #ok
Environment	: #document #config
Taxonomy	: #event #task #flow
Module	: { <i>action</i> : $\Sigma \rightarrow P \rightarrow P$ }
Process	: #process

```

record task      := (name: string)
record event     := (name: string)
record flow      := (source: task) (target: task)
record process
extends iterator :=
  (env: list iterator)
  (feed_id: N)
  (taxonomy: prod (prod (list task) (list flow)) (list event))

```

1.9.6 Управління процесами

$$\begin{array}{lcl}
 BPE & : & amend \quad \times _ \\
 & | & complete \quad \times _ \\
 & | & run \quad \times _
 \end{array}$$

```

record p      extends proto
record run    extends p
record amend  extends p
record complete extends p
definition bpe := run + amend + complete + ok + error + io

record proc
extends app bpe process :=
  (sup: list process)

```

1.9.7 Алгебра процесів

Алгебра процесів визначає базові операції мультиплексування двох чи декількох протоколів в рамках одного процесу (добуток), а також паралельного та повністю ізолюваного запуску включно зі стеком та областю пам'яті (сума) на віртуальній машині.

$$\begin{array}{lcl}
 \oplus & : & P \parallel P \\
 \otimes & : & P \mid P
 \end{array}$$

1.9.8 Лямбда числення

Для розробки нашої теорії ми застосуємо інтуїціоністську теорію типів Мартіна Льюфа. Ця система типів буде застосовуватися для доведення робочих теорем. База мови складає лямбда числення як внутрішня мова декартово-замкненої категорії. Лямбда числення в своїй основі складається з конструктора типу яка визначає функції та елімінатора – аплікації функції до аргументу.

$$\frac{\Gamma x : A \vdash B}{\Gamma \vdash \text{fun } x : A \rightarrow B} \qquad \frac{\Gamma f : A \rightarrow B \quad \Gamma a : A}{\Gamma \vdash f a : B}$$

1.9.9 Алгебраїчні типи даних

Далі йде внутрішня мова алгебраїчних типів даних яка складається з суми та добутку, за допомогою яких контруюються суми-протоколи та кортежі-пакети, а також **nil** типу-атому, яким зручно термінувати рекурсивні типи даних, такі як **cons**.

$$\frac{}{\Gamma \vdash \perp}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B} \qquad \frac{\Gamma x : A \times B}{\Gamma \vdash \text{fst } x : A; \Gamma \vdash \text{snd } x : B}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash a \mid b : A \otimes B} \qquad \frac{\Gamma x : A \otimes B}{\Gamma \vdash \text{inl } x : A; \Gamma \vdash \text{inr } x : B}$$

1.9.10 Процеси і протоколи

Також ми анонсуємо процес як фундаментальний тип даних, подібний до функції але який здатний тритати певний стан у вигляді типа коротежа та бути здатним реагувати на повідомлення суми.

$$\frac{\Gamma \vdash e : \text{Error} \quad \Gamma \vdash h : \text{History}}{\Gamma \vdash \text{module} : a : \Sigma \times P \rightarrow P} \quad \frac{\Gamma \vdash \text{env} : \text{Document} \quad \Gamma \vdash s : \text{Event}|\text{Task}|\text{Flow}}{\Gamma \vdash p : \text{Process}}$$

$$\frac{\Gamma \vdash \Sigma, X \quad \Gamma \vdash p : \text{Process}}{\Gamma \vdash \text{spawn}_{\Sigma} p : \pi}$$

$$\frac{\Gamma \vdash \text{spawn} : \pi \quad \Gamma \vdash m : \Sigma}{\Gamma \vdash \text{join}_{\Sigma} (m, \text{spawn}) \rightarrow \Sigma; \Gamma \vdash \text{async}_{\Sigma} (m, \text{spawn}) \rightarrow \Sigma}$$

1.9.11 Логіка та квантори

Далі йдуть квантори \forall та \exists які теж виражаються як конструкції типів:

$$\frac{\Gamma x : A \vdash B \quad \Gamma \vdash A}{\Gamma \vdash \Pi(x : A)B} \quad \frac{\Gamma \vdash a : A \quad \Gamma x : A \vdash B \quad \Gamma b : B(x = a)}{\Gamma \vdash (a, b) : \Pi(x : A)B}$$

$$\frac{\Gamma x : A \vdash B \quad \Gamma \vdash A}{\Gamma \vdash \Sigma(x : A)B} \quad \frac{\Gamma \vdash a : A \quad \Gamma x : A \vdash B \quad \Gamma b : B(x = a)}{\Gamma \vdash (a, b) : \Sigma(x : A)B}$$

$$\frac{\Gamma \vdash x : A \quad \Gamma \vdash x' : A}{\Gamma \vdash Id_A(x, x')}$$

рефлексивність	: $Id_A(a, a)$
підстановка	: $Id_A(a, a') \rightarrow B(x = a) \rightarrow B(x = a')$
симетричність	: $Id_A(a, b) \rightarrow Id_A(b, a)$
транзитивність	: $Id_A(a, b) \rightarrow Id_A(b, c) \rightarrow Id_A(a, c)$
конгруентність	: $(f : A \rightarrow B) \rightarrow Id_A(x, x') \rightarrow Id_B(f(x), f(x'))$

1.9.12 Контраваріантні процеси

Ко-процеси це моноїди з перевернутою стрілкою, які визначають зворотній шлях виконання події та обернену зміни стану в зворотньому порядку. З точки зору типів нічого не змінюється, однак оберненої функції-функтора процесу-моноїда може і не існувати. Процеси які є одночасно коваріантними та контраваріантними називаються інваріантними процесами, сигнатура яких алгебраїчно збігається з сигнатурою бінарного дерева.

$$\begin{array}{lcl} X & : & \times^i \\ \Lambda & : & \times^4 I X \Lambda_{next} \Lambda_{prev} \\ & | & \perp \end{array}$$

$st(p) = p(st - 1)$ — застосування функції до протокольної заявки та стану процесу
 $p - 1(st) = st - 1$ — обернена функція

1.9.13 Типи процесів

<i>action</i>	: $proc_{Proc} (Proc \mid \Sigma) \times process \rightarrow process$
<i>event</i>	: $proc_{App} (App \mid \Sigma) \times cx \rightarrow cx$
<i>operation</i>	: $proc_{Store} (Store \mid \Sigma) \times container \rightarrow container$

1.10 Віртуальна машина

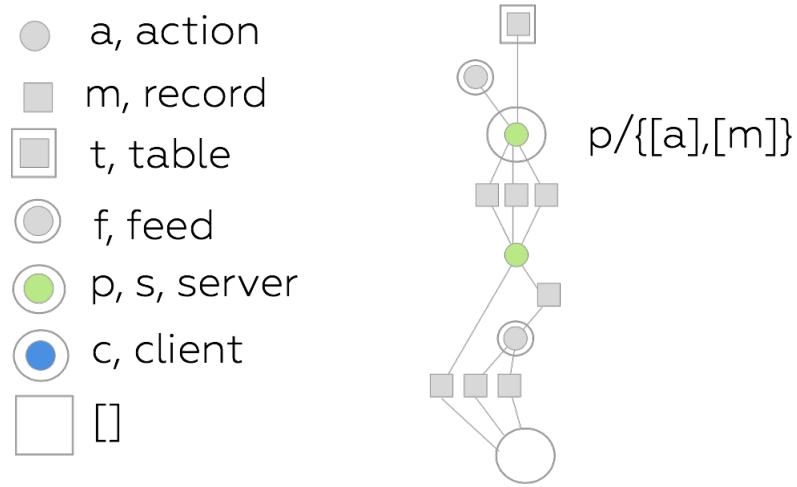


Рис. 1: Базові елементи віртуального середовища

action is the piece of code applied to message using pattern matching.

record is the tuple that defines a protocol.

client is the true customer of the system who is using service on daily basis.

server is the process nicely scheduled by the system.

feed is the server managed list of records.

table is the ixset or ets based queryable tables.

1.11 Сигнатури сервісів

app is the web clients monoidal state machine.

proc is the oriented graph monoidal executional engine.

store is the 2-category key-value database.

$$\begin{aligned}
 store^3 &: \times^3 \Lambda_{id_{seq}} \Lambda_{table} \Lambda_{feed} \\
 roster^2 &: \times^2 \Lambda_{feed} \Lambda_{feed}^2 \\
 app^2 &: \times^2 \Lambda_{session} \Lambda_{proc} \\
 depot^1 &: \times^1 \Lambda_{account} \\
 act^1 &: \times^1 \Lambda_{proc}
 \end{aligned}$$

1.12 Мета дослідження

Основною метою дослідження є забезпечення виконання критеріїв відповідності показників ефективності, детермінованості та якості шляхом впровадження результату даної теорії, комплексу програмного забезпечення у виробничий процес.

За рахунок компактності передбачається значна економія складності та простота у експлуатації, що веде до зниження собівартості впровадження. Коректність передбачає заміну автоматизованому тестуванню, а закритість середовища дозволяє гнучко використовувати середовище на різних виробничих платформах.

1.13 Результати

Одна з останніх систем, яка була впроваджена для заміни існуючої системи, мала показники у співвідношенні розміру 10, швидкості 20 та виконання 3. Інші системи у історичній послідовності мали ці показники такими 10, 2, 8, та 19, 5, 5. В середньому показники ефективності відрізняються більше ніж на пів порядку в кращу сторону після впровадження продуктів на базі розробленої моделі. Існують продукти також і на інших мовах програмування.

Додаток А. Персистентність послідовностей

Для забезпечення персистентного зберігання послідовностей використовується очевидний спосіб зберігання елементів послідовності у key-value сховищі разом з референсами на наступні, та у випадку контраваріантних процесів, також і на попередні елементи у послідовності. Таким чином ми можемо перелічити усі елементи послідовності. Акумулятор розділеної у просторі та часі операції накопичує усі необхідні агрегації після кожної операції з послідовністю. Таким чином похідна від алгебраїчної структури списку буде скаляр-акумулятор після завершення перелічення.

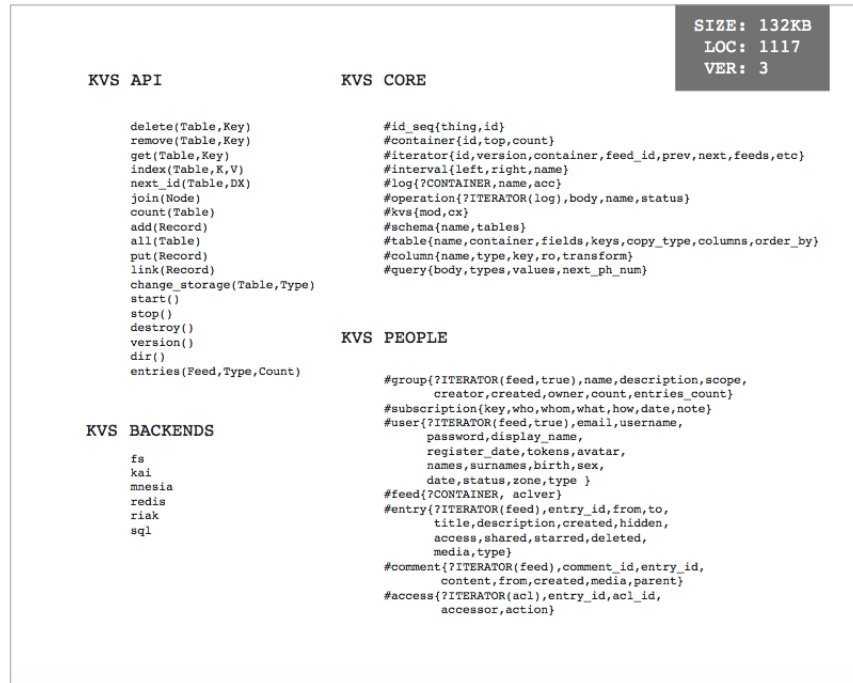


Рис. 2: Типи, функції та модулі

В даній роботі ми розглянемо формальні визначення сховища, його референсів, послідовностей через рекурсивні алгебраїчні типи даних та процеси, черги яких забезпечують впорядкованість послідовності запитів до послідовностей, що веде до лінійності операції у часі.

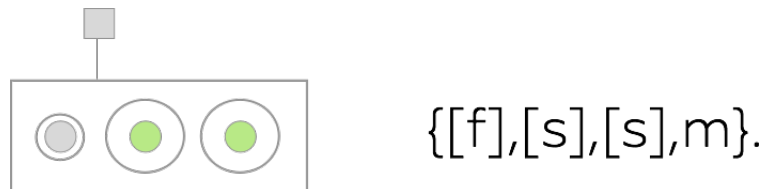


Рис. 3: Послідовності ключів, таблиць, індексів та інтерфейс

Додаток Б. Мультипротокольний чат

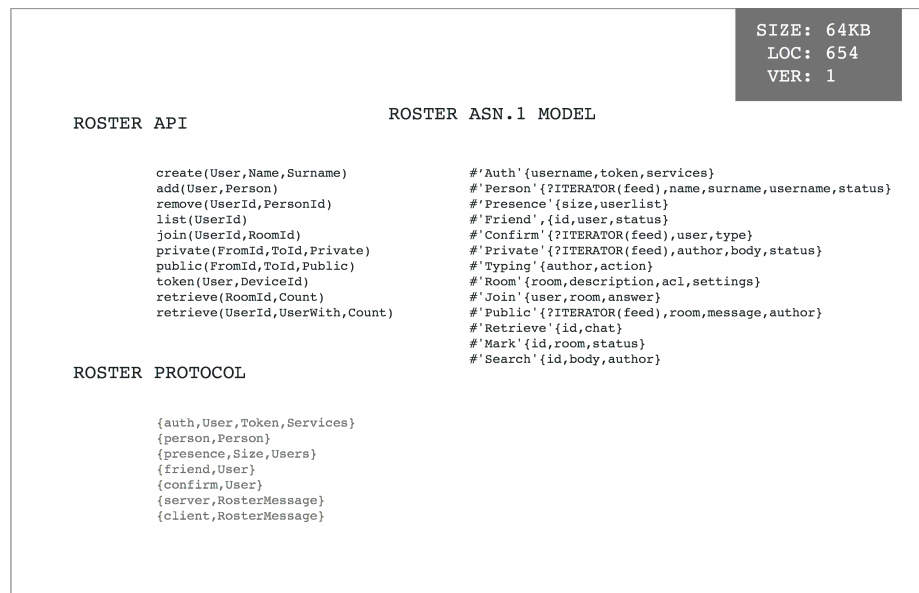


Рис. 4: Типи, функції та протокол

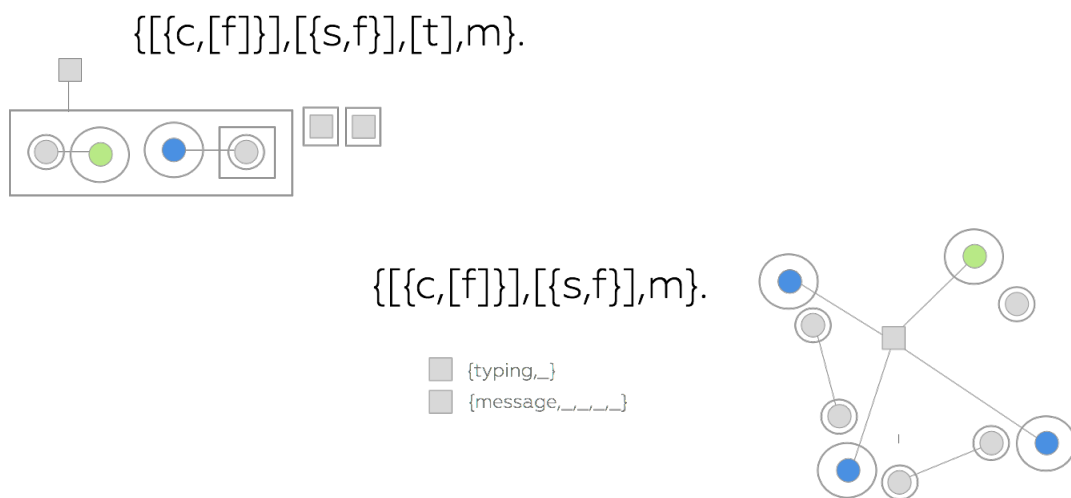


Рис. 5: Списки кімнат та користувачів і їхні чати, дві таблиці та протокол

Список литературы

- [1] Per Martin-Löf *Intuitionistic Type Theory*. 1984
- [2] J.Armstrong. *Making reliable distributed systems in the presence of software errors*. 2003.
- [3] Robin Milner. *A Calculus of Communicating Systems*. 1986.
- [4] William Lawvere. *Conceptual Mathematics*. 1997.
- [5] Benjamin Pierce. *Basic category theory for computer scientist*. 2004.
- [6] Сандерс Мак Лейн. *Категории для работающего математика*. 2004.
- [7] Leslie Lamport. *Specifying Systems*. 2004.
- [8] Michael Barr and Charles Wells. *Toposes, Triples and Theories*. 2000.
- [9] Michael Barr and Charles Wells. *Category Theory for Computing Science*. 1995.
- [10] И.Бакур, А.Деляну. *Введение в теорию категорий и функторов*. 1972.
- [11] Robin Milner. *Communicating and Mobile Systems: The π -calculus*. 1999.
- [12] Robin Milner. *The Polyadic π -Calculus: A Tutorial*. 1993.
- [13] Коваленко. *Теория Массового Обслуживания*. 1965.
- [14] Meseguer, Montanari. *Petri Nets Are Monoids*. 1990.
- [15] Philip Wadler *Call-by-Value is Dual to Call-by-Name*. 1000
- [16] D.Mostrous, V.Vasconcelos *Session Typing for a Featherweight Erlang*. 1990.
- [17] S.Marlow, P.Wadler *A practical subtyping system for Erlang*. 1997
- [18] A.Lindgren *A Prototype of a Soft Type System for Erlang*. 1996
- [19] C. McBride. *Dependently Typed Functional Programs and their Proofs*. 1999
- [20] C. McBride. *The Derivative of a Regular Type is its Type of One-Hole Contexts*.
- [21] C. Schwarzweller. *Mizar Verification of Generic Algebraic Algorithms*. 1997
- [22] L.Moura., S.Kong *Elaboration in Dependent Type Theory*. 1997
- [23] T.Coquand, G.Huet *The Calculus of Constructions*. 1988
- [24] L.Lamport *Paxos Made Simple*. 2001
- [25] A.Chlipala *Certified Programming with Dependent Types* 2015
- [26] H.Gueves, R.Pollak, F.Wiedijk, J.Zwanenburg *A Constructive Algebraic Hierarchy in Coq* 2002

- [27] E.Brady *Programming in IDRIIS: A Tutorial* 2015
- [28] S.Kaes <http://tuprints.ulb.tu-darmstadt.de/epda/000544/diss.pdf> 2005
- [29] A.Baer *Programming with Algebraic Effects and Handlers* 2012
- [30] R.Harper *Practical Foundations for Programming Languages* 2015