synrc research center s.r.o.
Roháčova 141/18, Praha 3 13000, Czech Republic

Системна інженерія та верифікація уніфікованого обчислювального середовища

System Engineering and Verification of Unified Execution Environment

Павло Маслянко, Київський Політехнічний Інститут

Максим Сохацький, Synrc Research Center

Листопад 2015

Содержание

1 Вступ

1.1 Системна інженерія

Протягом історії обчислювальної техніки було створено різні класи та способи обчислень, різні тоеорії 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 ізольовану по відношенню до операційної системи систему типів разом з моделью акторів, що дозволяє самостійно розподіляти час обчислювального середевища. Стан таких віртуальних апаратно незалежний та може бути перенесений на віртуальні машини які працюють на процесорах інших архітектур. Інший класс 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 унфіковані системи які здатні виконуватися без операційної системи існують на трьох платформах (на OCaml це MirageOS, на Haskell це HaLVM, для Erlang це віртуальна машина LING, яка створена в Україні). В даній роботі дається обгрунтуванню вибору віртуальної машини Erlang та її операційної семантки 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 якогомога ближче до аппаратного 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 сучаних 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 разподілених у просторі та часі систем (приклали таких систем: NuPRL від університету Корнела в Нью-Йорку; TLA+ фреймворк Леслі Лампорта з Microsoft Research, за який він отримав премію Тюрінга), а також системи з залежними типами (теорія типів Мартіна Льофа) побудовані на основі числення індуктивних конструкцій (приклади таких систем:
Coq побудована на мові OCaml від національного науково-дослідного інституту Франції INRIA; Agda побудовані на мові Haskell від швецького інституту технологій Чалмерс; Lean побудована на мові C++ від Microsoft Research та Університету Каргені-Мелона; Idris подудована на мові Haskell Едвіна Бреді з шотландського Університету ім. св. Андрія; F* – окремий проект Microsoft Research).

1.4 Компактні адаптивні протоколи

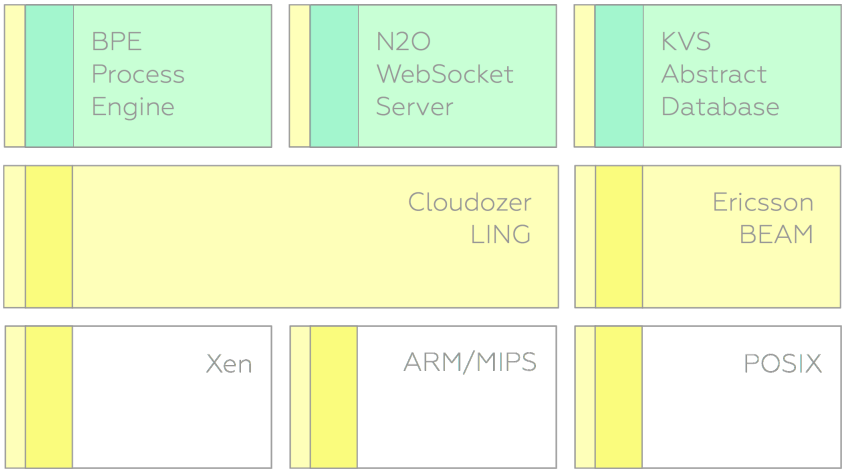
Зараз на планеті налічується близько 50 міліардів пристроїв які повинні обслуговуватися масштабованими ефективними серверними комплексами які гарантовано не виконують додаткових та непотрібних обчислень. Це потребує повного перепроектування усіх протоколів які виникають між компонентнами та підсистемами. Ця работа 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 залежнити 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 системо 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 аппаратне 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 можно верифікувати виконання команд віртуальної машини та компіляцію цих команд в байт-код.

Сервіси Верифікована система типів верхнього рівня моделі OSI — це центральний фокус даногої роботи. Ми розглянемо декілька основних сервісів які запезпечують персистентність, виконаня 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 констистентності операційних логів розподілених базах данних.

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 найкомактніших 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 данних які ефективно використоують розподілені у просторі та часі ресурси, як то об'єми памяті та обчислювальні потужності. Принципи та підходи паралельного та узгодженого програмування дають змогу масштабувати системи та обчисленя, однак анонсують нові теорії для забезпечення коректності в умавах підвищеної складності алгоритмів у розподілених системах, такі як алгоритми забезпечення консистентності та транзакційності у розподілених системах PAXOS [23] та CR [37]. Сучасні обчислювальні середовища повинні ефективно управляти великими масивами даних та необмежено масштабуватися. Розподілена архтітектура сервера транзакцій які працюють поверх формального верифікованого сховища даних буде представлена окремою роботою. Даний прототип існує у вигляді координатора транзакцій який працью 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 формалізовної віртуальної машини достатньо визначити систему типів команд мови процессор 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 лінгвістистичного забезпечення у якості формальних методів, ми будемо формувати формальну теорію та бібліотеку типів для формальної реалізації ключових підсистем.

Усі існуючі уніфіковоані системи, оскільки є замкеними, пропонують повністю свій інтерфейс взаємодії, свою операціїну семантку та свою систему типів. У цьому смислі 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 компонена 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 формулюються 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 основане на віртуальних машинах зі своєю системою типів та байткодом, які уже використовуються у промисловій експлуатації в телекомунікайній сфері та є лідерами у цій області – це телекомукаційна платформа від Ericsson – Erlang/OTP. Ступінь проникнення цієї технологій 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 міліардів повідомлень в день у порівнянні з SMS трафіком, який генерується щоденно у розмірі 20 міліардів 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 виконнання, забезпечуючи при цьому семантику, яка використовуються 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 управліннями процессами 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 будуємо 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 властівості 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 застововується 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 транслюються 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 базовії теорії доведеться формально описани 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 комплесної теорії верифікованих середовищ послідовних обчислень розподілених узгоджених паралельних систем.

1.7.1 Теорія масового обслуговування

Теорія массового 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 пропускної 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 Exe та категоріальна семантика числення процесів. Можна було би використовувати абстракну 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 коррекності усіх нормалізованих підсистем. На основі теорії Леслі Лампорта [6], 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 непротиречивості та неблокованості у синхронному виконанні мобільних процесів. Так як сучасний Web можно 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 Fω системах Haskell та Scala. Наша результуюча система типів буде тегована система типів мови Erlang.

1.8.3 Інтуітіоністична теорія типів Мартіна Льофа

Системи з залежними типами як верифікаційні математичні формальні моделі для доведення корректності. 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 [29]. Однак Agda та Idris використовують для побудови алгебраїчної теорії типи класів, а у Idris взагалі відсутні поліморфірні індуктивні структури та коіндуктивні структури. В Lean теж відсутні коіндуктивні структури проте повністю реалізована теорія HoTT на нерекурсивних поліморфних структурах що обєднує основні чотири класи математичних теорій: логіка, топологія, теорія множин, теорія типів. Як було показано Стефаном Касом [30], 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 використаннями 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 метамоделей 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 метамоделі 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 аппаратного 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 застовувати алгебраїчний підхід до генералізації теорії процесів. З алгебраїчної точки зору процесси, або кінечні автомати, представляють собою напівгрупи дій. З категоріальної точки зору процесси — це функтори які перетворюють категорії з декартовими добутками типів протоколу та стану процеса на категорію станів. Кожен процес визначається таким функтором, адже простір дій Σ для кожного процесу є унікальний.

$$\begin{array}{lcll} A & : & \Sigma \times X \to X & \\ X & : & \Sigma \times \Lambda_X & \\ & | & \bot & \\ \Sigma & : & ok & \times \_ \\ & | & error & \times \_ \\ & | & io & \times \_ \times \_ \end{array}$$

У функціональних мовах програмування така категорія будем мати вигляд програми, де функтор буде представлений фунцією 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.

$$\begin{array}{lcl} spawn & : & \top \to S \\ run & : & S \to \bot \\ \bot & : & \odot S \to S \\ complete & : & \otimes S \to S \end{array}$$

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

```
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 процессів 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 [13] мережі петрі, а значить і процеси є моноїдами які генерують лінійну послідовність подій.

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

Саме лінійність кінцевого процессу 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 єфектів у вигляді списків, з огляду на лінійність процесу обчислень. Так в мові Idris ефекти середовища дійсно представлені у вигляді списків. Як показано [27], ефекти навіть вносяться як мовні елементи у функціональну мову, як це наприклад існує в імперативних мовах у вигляді типів виняткових ситуації.

Класично до ефектів, крім помилок та виключень, відносять також усі операції що взяємодіють зі змінними за межами локального контексту виконання функції (параметри чи локальні змінні в стеку функції). Також до ефектів відносят стан процессу 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 можно 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 ефектими які виникають в процесі обчислень. І в розподілених системах зручно явно оперувати такими ланцюгами ефектів. Тому ми описуємо універсальних 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 оперує обчисленнями як послідовностями, і може їх склеювати, підрізати, пробігати та згортати.

Ми почнемо з двох найголовніших послідовностей *ok* (послідовніст 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.

$$
\begin{array}{rcll}
KVS & : & get & \times \_ \\
 & : & put & \times \_ \\
 & | & add & \times \_ \\
 & | & remove & \times \_ \\
 & | & cut & \times \_ \\
 & | & append & \times \_ \\
 & | & fold & \times \_
\end{array}
$$

```
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 визначини процес ми спочатку визначимо домени або типи, які фігурують при визначенні процесу.

$$
\begin{array}{lcl}
\text{Errors} & : & \#\text{error} \\
\text{Sucesses} & : & \#\text{ok} \\
\text{Environment} & : & \#\text{document} \mid \#\text{config} \\
\text{Taxonomy} & : & \#\text{event} \mid \#\text{task} \mid \#\text{flow} \\
\text{Module} & : & \{action : \Sigma \to P \to P\} \\
\hline
\text{Process} & : & \#\text{process}
\end{array}
$$

```
 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

$$
\begin{array}{rcll}
BPE & : & amend & \times \_ \\
 & \mid & complete & \times \_ \\
 & \mid & run & \times \_
\end{array}
$$

```
 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 ізольованого запуску включно зі стеком та областю памяті (сума) на віртуальній машині.

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

1.9.8 Лямбда числення

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

$$\frac{\Gamma\ x : A \vdash\ B}{\Gamma \vdash fun\ x : A \to B} \qquad \frac{\Gamma\ f : A \to B \qquad \Gamma\ a : A}{\Gamma \vdash f\ a : B}$$

1.9.9 Алегбраїчні 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**.

$$\frac{}{\Gamma \vdash\ \bot}$$

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

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

1.9.10 Процеси і протоколи

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

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

$$\frac{\Gamma\ \vdash \Sigma, X \qquad \Gamma\ \vdash p : Process}{\Gamma \vdash spawn_{\Sigma}\ p : \pi}$$

$$\frac{\Gamma\ \vdash spawn : \pi \qquad \Gamma\ \vdash m : \Sigma}{\Gamma \vdash join_{\Sigma}\ (m, spawn) \to \Sigma; \Gamma \vdash async_{\Sigma}\ (m, spawn) \to \Sigma}$$

1.9.11 Логіка та квантори

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

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

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

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

$$\begin{array}{lcl}
\text{рефлексивність} & : & Id_A(a, a) \\
\text{підстановка} & : & Id_A(a, a') \to B(x = a) \to B(x = a') \\
\text{симетричність} & : & Id_A(a, b) \to Id_A(b, a) \\
\text{транзитивність} & : & Id_A(a, b) \to Id_A(b, c) \to Id_A(a, c) \\
\text{конгруентність} & : & (f : A \to B) \to Id_A(x, x') \to Id_B(f(x), f(x'))
\end{array}$$

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 зоро типів нічого не змінюється, однак оберненої функції-фунтора процесу-моноїда може і не існувати. Процеси які є одночасно коваріантними та контраваріантними називаються інваріантними процесами, сигнатура яких алгебраїчно збігається з сигнатурою бінарного дерева.

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

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

1.9.13 Типи процесів

$$\begin{array}{lcl}
action & : & proc_{Proc}\ (Proc \mid \Sigma) \times process \to process \\
event & : & proc_{App}\ (App \mid \Sigma) \times cx \to cx \\
operation & : & proc_{Store}\ (Store \mid \Sigma) \times container \to container
\end{array}$$

1.10 Сигнатури сервісів

app — веб сервер додатків, та комутатор протоколів
proc — дводольний граф обчислень для бізнес-процесів
store — сховище ланцюгових послідовностей

Веб сервер додатків у своєму контексті зберігає різноманітні таблиці для різного роду даних (cache, cookies, session), а також управляє процесами користувачів, до яких підключаються певні протоколи в залежності від запитів користувачів. Тому у його сигнатурі ми побачимо два списки: список таблиць та список процесів.

Для сховища лінійних послідовностей сигнатура сервсів буде включати список усіх таблиць сховища де зберігаюьтся кортежі елементів послідовносетй, їхні генератори унільних ідентифікаторій, а також список усіх унікальних послідовностей.

Для бізнес процесів сигнатура сервіса найпростіша: це просто список бізнес процесів, для кождого з яких тримається послідовність ефектів (feed) у сховищі kvs.

$$\begin{array}{lcl} store^3 & : & \times^3\Lambda_{id_seq}\ \Lambda_{table}\ \Lambda_{feed} \\ app^2 & : & \times^2\Lambda_{tables}\ \Lambda_{procs} \\ act^1 & : & \times^1\Lambda_{procs} \end{array}$$

1.11 Віртуальна машина

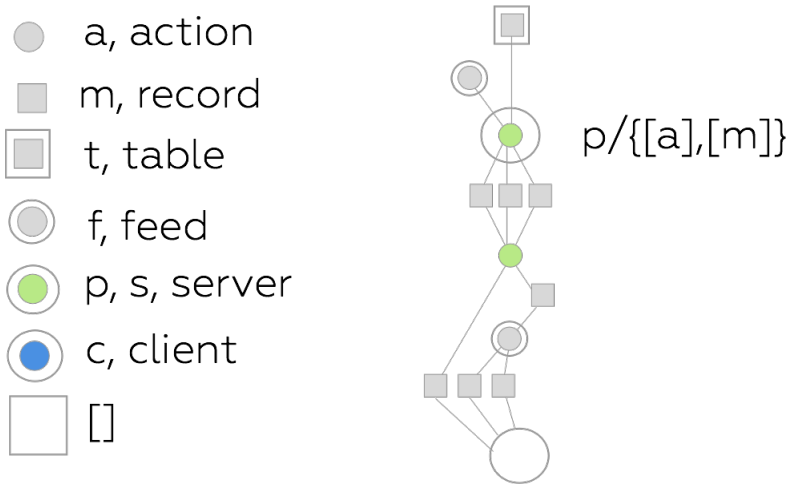


Рис. 1: Базові елементи віртуального середовища

action — функція яка застосовується до протокольного повідомлення
record — кортеж у якому кодується протокольне повідомлення
client — користувач системи представлений у вигляді процеса
server — процес який обслуговує клієнтів та реагує на протокол
feed — збережена послідовність обчислень
table — масив кортежів з B-tree індексами

1.12 Мета дослідження

Основною метою дослідження є збезпечення 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 сниження собівартості впровадження. Коректність передбачає заміну автоматозаваному тестуванню, а закритість середовища дозволяє гнучко використовувати середовище на різних виробничих платформах.

1.13 Результати

Результати будут представлені з точки зору: показників ефективності на прикладі розміру бібліотек та результатах впровадження; показники швидкодії у вигляді графіку навантаження, що показує ємність системи; та показники швидкодії персистентного сховища у застосуванні до розподілених систем на прикладі розподіленої бази даних, що працює за протоколом ланцюгової реплікації.

- Показники ефективності
- Застосування у телекомунікаційних додатках
- Застосування у розподілених системах

Показники ефективності

У виробництві, після коректності, розмір є одним з найважливіших чинників. Адже ціна обслуговування, та загальна складність системи зі зменшенням обсягу продуку зменшується, що дає змогу більш оперативно вносити зміни до системи, а також зменшує час для адаптації операторам системи. Крім того, програмістам набагато комфортніше працювати з малими об'ємами коду да добре структурованими та компактними проектами. Покажемо у цифрах переваги нашої системи для мови Erlang:

Бібліотека	LOC	Size,KB	Опис
LING	64K	1800	Віртуальна Машина на мові C
PIE	5.6K	350	Emacs-подібний текстовий редактор
CR	2.2K	156	Chain Replication база даних з XA протоколом
N2O[1]	1.2K	124	Мультиплікатор протоколів та веб додатки
MAD	980	116	rebar.config сумісний менеджер пакетів
KVS[2]	1.2K	92	Персистентність ефектів
BPE[3]	650	72	Система управління бізнес-процесами BPMN 2.0
LDAP	730	64	LDAP сервер
MQS	300	28	AMQP клієнт

Застосування у телекомунікаційних додатках

Щодо аргументації швидкості, та вибору платформи, ми пропонуємо велику конгрегацію веб фреймворків сучасних функціональних та імперативних мов програмування: Сюди увійшли Haskell (Warp), OCaml (OCsigen), Erlang (Cowboy/N2O), Clojure (http-kit), JavaScipt (Node), Go (gohttp), D (Vibe). Як видно з графіка у вищий клас попали Haskell, Java та Erlang HTTP стеки. Тестування проводилося за допомогою утіліти **wrk** для HTTP трафіку, та **tcpkali** для WebSocket трафіку. Тому можно з впевненістю сказати, що Erlang якнайкраще підходить у тому числі для веб додатків, завдяки своїм телекомунікаційним властивостям. Крім того завдяки відсутності непотрібних прошарків у системи можно побачити, що N2O мінімально впливає на пропускну спроможність тракту.

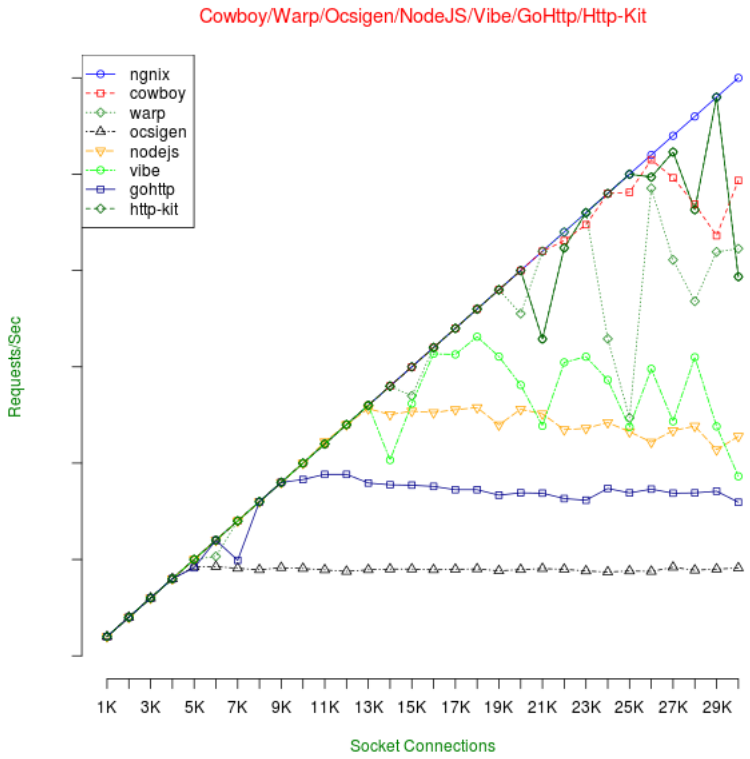


Рис. 2: Навантаження яке витримує N2O

Спосіб представлення функціональності веб-додатків у вигляді модуля який представляє єдину функцію напівгрупи активностей та реагує на унікальний протокол повідомлень добре зарекомендував себе у виробництві, так система з 18 протоколами, 96 формами та 24 зовнішніми сервісами які мають 55 точок підключення була впроваджена протягом одного року відділом з трьох програмістів. Система завдяки сховищу KVS підтриує необмежені лінійні (впорядковані у часі) ідентифікатори, версії кортежів та протокольних повідомлень, різні сховища нижчого рівня, у тому числі і стандартне сховище Erlang – **mnesia**.

Застосування у розподілених системах

Що стосується швидкості системи сховища ефектів, то тут хочеться представити імплементацію транзакційного сховища CR [37] з послідовностями KVS у якості транзакційного журналу. У якості алгоритму оракула був вибраний алгоритм RAFT [38]. Як відомо, алгоритм ланцюгової реплікації помножує латенсі усіх нод, на відміну від бродкаст алгоритмів типу PAXOS, однак значно економить міжпротокольний трафік та зберігає простоту алгоритму. Ми реалізували алгоритм ланцюгової реплікації на базі KVS в якості операційного журналу, та отримали наступні результати.
Для трьох машин у локальній мережі ми маємо середнє значення latency у розмірі 100мс при реплікації на три машини і 300мс максимальне значення для бази даних **mnesia** у конфігурації **disc_copies**.

vnode	i	n	top	log	latency
1217918031109085765169737360596902516379943785 81	1	1	6506	1607	1/315/97
2435836062218171530339474721193805032759887571 62	2	1	6508	1662	1/317/100
3653754093327257295509212081790707541398313574 3	3	1	6510	1658	2/317/105
4871672124436343060678949442387610065519775143 24	4	1	6505	1583	1/317/104
6089590155545428825848686802984512581899718929 05	5	2	6499	1637	3/317/115
7307508186654514591018424163581415098279662714 86	6	2	6510	1664	2/318/117
8525426217763600356188161524178317614659606500 67	7	2	6501	1634	2/311/115
9743344248872686121357898884775220131039550286 48	8	2	6500	1575	3/290/96
1096126227998177188652763624537212264741949407 229	9	3	6497	1607	3/316/118
1217918031109085765169737360596902516379943785 810	10	3	6510	1662	3/318/117
1339709834219994341686711096656592768017938164 391	11	3	6496	1658	3/311/106
1461501637330902918203684832716283019655932542 972	12	3	6505	1583	2/295/104

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.

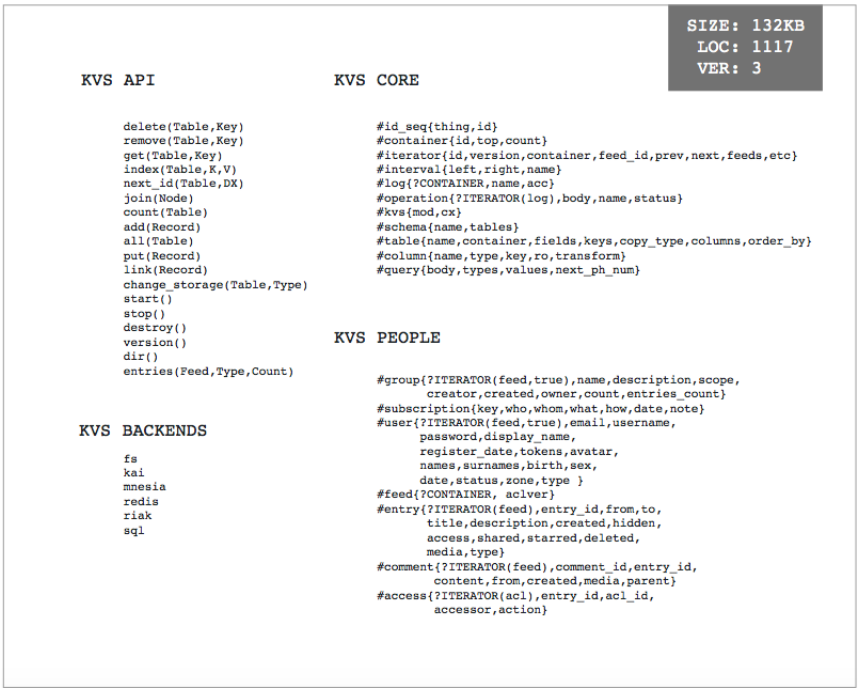


Рис. 3: Типи, функції та модулі

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.

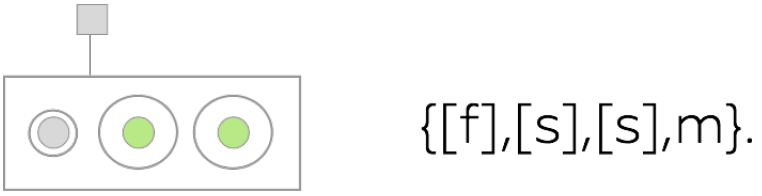


Рис. 4: Послідовності ключів, таблиць, індексів та інтерфейс

Додаток Б. Мультипротокольный чат

Рис. 5: Типи, функції та протокол

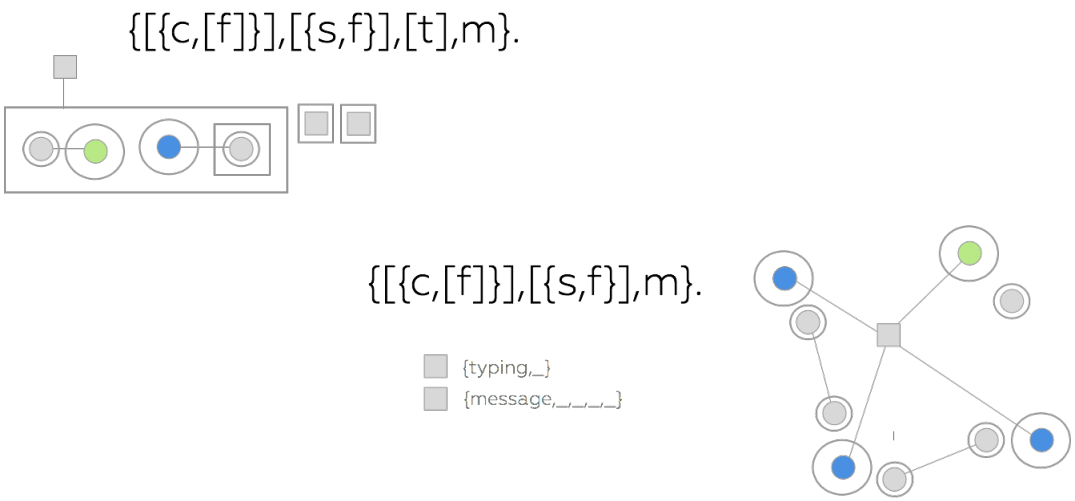


Рис. 6: Списки кімнат та користувачів і їхні чати, дві таблиці та протокол

Список литературы

[1] Per Martin-Löf *Intuitionistic Type Theory.* 1984

[2] Robin Milner. *A Calculus of Communicating Systems.* 1986.

[3] William Lawvere. *Conceptual Mathematics.* 1997.

[4] Benjamin Pierce. *Basic category theory for computer scientist.* 2004.

[5] Сандерс Мак Лейн. *Категории для работающего математика.* 2004.

[6] Leslie Lamport. *Specifying Systems.* 2004.

[7] Michael Barr and Charles Wells. *Toposes, Triples and Theories.* 2000.

[8] Michael Barr and Charles Wells. *Category Theory for Computing Science.* 1995.

[9] И.Бакур, А.Деляну. *Введение в теорию категори и функторов.* 1972.

[10] Robin Milner. *Communicating and Mobile Systems: The π-calculus.* 1999.

[11] Robin Milner. *The Polyadic π-Calculus: A Tutorial.* 1993.

[12] Коваленко. *Теория Массового Обслуживания.* 1965.

[13] Meseguer, Montanari. *Petri Nets Are Monoids.* 1990.

[14] Philip Wadler *Call-by-Value is Dual to Call-by-Name.* 1000

[15] D.Mostrous, V.Vasconcelos *Session Typing for a Featherweight Erlang.* 1990.

[16] S.Marlow, P.Wadler *A practical subtyping system for Erlang.* 1997

[17] A.Lindgren *A Prototype of a Soft Type System for Erlang.* 1996

[18] C. McBride. *Dependently Typed Func- tional Programs and their Proofs.* 1999

[19] C. McBride. *The Derivative of a Regular Type is its Type of One-Hole Contexts.*

[20] C. Schwarzweller. *Mizar Veri cation of Generic Algebraic Algorithms.* 1997

[21] L.Moura., S.Kong *Elaboration in Dependent Type Theory.* 1997

[22] T.Coquand, G.Huet *The Calculus of Constructions.* 1988

[23] L.Lamport *Paxos Made Simple.* 2001

[24] R.Harper *Practical Foundations for Programming Languages* 2015

[25] A.Chlipala *Certified Programming with Dependent Types* 2015

[26] E.Brady *Programming in IDRIS: A Tutorial* 2015

[27] A.Baer *Programming with Algebraic Effects and Handlers* 2012

[28] A.Chlipala *An Introduction to Programming and Proving with Dependent Types in Coq* 2010

[29] H.Gueves, R.Pollak, F.Wiedijk, J.Zwanenburg *A Constructive Algebraic Hierarchy in Coq* 2002

[30] S.Kaes *http://tuprints.ulb.tu-darmstadt.de/epda/000544/diss.pdf* 2005

[31] A.Jeffrey, J.Rathke *The Lax Braided Structure of Streaming I/O* 1998

[32] A.Jeffrey *Linear-time Temporal Logic Propositions as Types Proofs as Functional Reactive Programs* 2012

[33] P.Wadler *Propositions as Types* 2014

[34] N.Benton, J.Hughes,E.Moggi *Monads and Effects* 2002

[35] C.McBride, R.Patterson *Applicative programming with effects* 2002

[36] P.Wadler *Monads for functional programming*

[37] H.Abu-Libdeh, R.Renesse, Y.Vigfusson *Leveraging Sharding in the Design of Scalable Replication Protocols*

[38] R.Renesse, C.Ho, N.Schiper *Byzantine Chain Replication*

[39] R.Renesse, N.Schiper *Chain Replication for Supporting High Throughput and Availability*

[40] J.Armstrong. *Making reliable distributed systems in the presence of sodware errors.* 2003.