Міністерство освіти і науки України
Національний технічний університет України ”Київський політехнічний інститут імені Ігоря Сікорського”

Кваліфікаційна наукова праця
на правах рукопису

Сохацький Максим Еротейович

УДК 510.2—510.6

Дисертація

Концептуальна модель системи доведення теорем на основі гомотопічної теорії типів

Спеціальність 01.05.03 — математичне та програмне забезпечення обчислювальних машин і систем

Подається на здобуття наукового ступеня доктора філософії. Дисертація містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на джерела.

Науковий консультант: Павло Павлович Маслянко

Київ — 2018

УДК 510.2

Максим Сохацький

Мова простору

15 жовтня 2018 e8020d8141ece25695efc185aec732f35867386b

11 живтня 2018 64c2e49765d3b997d68999c88448fef7136079d4

http://bit.ly/groupoid

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

CoC — Calculus of Constructions

CiC — Calculus of Inductive Constructions

PTS — Pure Type System

MLTT — Martin-Lof Type Theory

ITS — Inductive Type System

HTS — Homotopy Type System

CPS — Continuation Passing Style

AST — Abstract Syntax Tree

ВСТУП

Присвячується Маші та Міші

Тут дамо тему, предмет та мету роботи, а також дамо опис структури роботи.

Актуальність роботи

Ціна помилок в індустрії надзвичайно велика. Наведемо відомі приклади: 1) Mars Climate Orbiter (1998), помилка[1] невідповідності типів британської метричної системи, коштувала 80 мільйонів фунтів стерлінгів. Невдача стала причиною переходу NASA[2] повністю на метричну систему в 2007 році. 2) Ariane Rocket (1996), причина[3] катастрофи – округлення 64-бітного дійсного числа до 16-бітного. Втрачені кошти на побудову ракети та запуск 500 мільйонів доларів. 3) Помилка в FPU в перших Pentium (1994), збитки на 300 мільйонів доларів. 4)

[1] Mars Climate Orbiter Mishap Investigation Board Phase I Report November 10, 1999. `https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf`

[2] National Aeronautics and Space Administration, національна адміністрація аеронавтики та космосу США

[3] ARIANE 5 Flight 501 Failure, `http://www-users.math.umn.edu/~arnold/disasters/ariane5rep.html`

Помилка[4] в SSL (heartbleed), оцінені збитки у розмірі 400 мільйонів доларів. 5) Помилка у логіці бізнес-контрактів EVM (неконтрольована рекурсія), збитки 50 мільйонів, що привело до появи верифікаторів та валідаторів контрактів[5], [6]. Більше того, і найголовніше, помилки у програмному забезпеченні можуть коштувати життя людей.

Формалізована постановка задачі

Тут стисло розкривається об'єкт, предтем, мета та завдання дослідження, та виділяються основні методи, які будуть використовуватися в роботі.

Об'єкт та предмет дослідження. Об'єктом дослідження в широкому сенсі є множина всіх формальних мов, та можливих зв'язків між ними.

Формально, об'єктом дослідження данної роботи є: 1) системи верифікації програмного забезпечення; 2) системи доведення теорем; 3) мови програмування; 4) операційні системи, які виконують обчислення в реальному часі; 3) їх поєднання, побудова формальної системи для уніфікованого середовища, яке поєднує середовище виконання та систему верифікації у єдину систему мов та засобів.

[4] The Matter of Heartbleed.
`http://mdbailey.ece.illinois.edu/publications/imc14-heartbleed.pdf`

[5] Vandal: A Scalable Security Analysis Framework for Smart Contracts,
`https://arxiv.org/pdf/1809.03981.pdf`

[6] Short Paper: Formal Verification of Smart Contracts,
`https://www.cs.umd.edu/~aseem/solidetherplas.pdf`

Предметом та методом дослідження такої системи мов є теорія типів, як сучасний фундамент математики, який стисло та компактно представляє не тільки теорію множин, але і теорію категорій, алгебраїчну топологію та дифференціальну геометрію. Теорія типів вивчає обчислювальні властивості мов та виділилася в окрему науку Пером Мартіном-Льофом як запит на вакантне місце у трикутнику теорій, які відповідають ізоморфізму Каррі-Говарда-Ламбека (Логіки, Мови, Категорії).

Мета і завдання дослідження. Одна з причин низького рівня впровадження у виробництво систем верифікації – це висока складність таких систем. Складні системи верифікуються складно. Ми хочемо запропонувати спрощений підхід до верифікації — оснований на концепції компактних та простих мовних ядер для створення специфікацій, моделей, перевірки моделей, доведення теорем у теорії типів з кванторами.

Метою цього дослідження є побудова єдиної системи, яка поєднує середовище викодання та систему верифікації програмного забезпечення. Це прикладне дослідження, яке є сплавом фундаментальної математики та інженерних систем з формальними методами верицікації.

Головним завданням цього дослідження є побудова мінімальної системи мовних засобів для побудови ефективного циклу верифікації програмного забезпечення та доведення теорем. Основні компоненти системи, як продукт дослідження: 1) інтерпретатор безтипового лямбда числення; 2) компактне ядро — система з однією аксіомою; 3) мова з індуктивними типами; 4) мова з гомотопічним інтервалом $[0, 1]$; 5) уніфікована базова бібліотека; 6) бібліотека математичних компонент.

Методи дослідження. Існує багадо підходів для формальної специфікації, верифікації та валідації, усі вони даються у розділі 1 та бгрунтувується вибір методу моделювання з використанням мови з залежними типами (теорії типів Мартіна-Льофа). Для разкриття семантки цього методу використовується категорний метод та категоріальна логіка – теорія топосів. Теорія категорій довела свою корисність не тільки для математики[7], але і для програмного забезпечення[8]

Наукова новизна

Автор спробував поглянути на проблему розширення, або мовного доповнення топосів та категорій де відбуваються обчислення, з точки зору теорії типів, та їх мовних синтаксисів та категорій. За допомогою спектрального розкладення на елементарні мови, які репрезентують певні типи та описуються сигнатурами ізоморфізмів, будується єдиний погляд на еволюцію мови та її покомпонентний аналіз.

В рамках розробленого фреймворку будується архітектура системи доведення теорем та обчислювального середовища (яке складається з верифікованого засобами Rust інтерпретатора та операційної системи), яке побудовано за сучасними стандартами: 1) відсутність системи

[7]Близько 10 робіт медалістів премії Філдса грунтуються на категорних методах

[8]Категорні бібліотеки таких мов як Haskell, Scala тощо.

управління пам’яті у реальному часі (тільки на стадії компіляції); 2) автоматична векторизація за допомогою AVX інструкцій; 3) дані ніколи не копіюються; 4) в системі немає очікування, полінгу та даремного витрачання ресурсів.

Перший кубічний верифікатор, в якому аксіома унівалентності Воєводського має конструктивну інтерпретацію, це cubicaltt авторства Андерса Мортберга (2017). Гомотопічні системи типів та системи типів в зв’язаних топосах є сучасним поглядом на конструктивну математику на шляху до формалізації таких моделей як інфінітезімальні околи, нескінченно малі (які потрібні для моделювання послідовності Коші) тощо.

Також це перша глобальна робота по створенню базової кубічної бібліотеки на базі розробленої концепції.

Практичні результати

В результаті цього дослідження встановлено, що формалізація системи програмування та доведення теорем можлива в рамках одніє уніфікованої системи, та показано на прикладі спектр мовних засобі [починаючи з інтерпретатора та безтипового лямбда числення, операційної системи середовища виконання (розділ 2), а також гомотопічної базової бібліотеки (розіл 3), здатної охопити майже всь глибину математики (розділ 4)] необхідних для побудови замкненої формальної системи доведення теорем та виконання програм.

Основные результаты

Автор особисто створив інтерпретатор, декілька верифікаторів та базову бібліотеку як приклади використання та моделювання системи доведення теорем. Автор також розробив курс гомотопічної теорії типів, на якому здійнюється формалізація певних розділів математики (теорія категорій, різні частини алгебраїчної топології та диференціальної геометрії). Окрім того створений сайт, присвячений документації по базовій бібліотеці для кубічного верифікатора гомотопічної мови програмування. Також частина моделей розроблений автором попала в апстрім кубічного верифікатора, як приклади використання.

Апробація результатів роботи

Відбулися виступи на двох конференціях: MMCTSE, IAI.

Структура роботи та публікації

Якщо коротко суть роботи зводиться до побудови системи, яка складається з: i) середовища виконання; ii) формального інтерпретатора; iii) системи формальних мов для доведення теорем математики, програмної інженерії та філософії.

Формальна верифікація. У розділі 1 дається огляд існуючих рішень для доведення властивостей систем та моделей, класифікуються мови програмування та системи доведення теорем.

Концептуальна модель системи доведення теорем на основі гомотопічної теорії типів. У розділі 2 розглядається повний стек формального програмного забезпечення від віртуальної машини, байт-код інтерпретатора та середовища виконання та планування процесів до формальної мови для доведення теорем (або сімейства мов).

Базова бібліотека. У розділі 3 описується базова бібліотека, написана на самій потужній формальній мові системи доведення теорем.

Математичні компоненти. У розділі 4 пропонується ряд математичних моделей та теорій з використанням базової бібліотеки розділу 3 та мови гомотопічної системи типів.

РОЗДІЛ 1
ФОРМАЛЬНА ВЕРИФІКАЦІЯ

Перша глава дає огляд існуючих рішень та вступ до предмету формальної верифікації. Розглядається класифікація систем верифікації, систем доведення теорем, та систем моделювання. Зображається місце дослідження у області та дотичні системи, такі як формалізовані середовища виконання.

1.1. Формальна верифікація та валідація

Для унеможливлення помилок на виробництві застосовуються різні методи формальної верифікації. Формальна верифікація — доказ, або заперечення відповідності системи у відношенні до певної формальної специфікації або характеристики, із використанням формальних методів математики.

Дамо основні визначення згідно з міжнародними нормами (IEEE, ANSI)[1] та у відповідності до вимог Європейського Аерокосмічного Агенства[2]. У відповідності до промислового процессу розробки, верифікація та валідація програмного забезпечення є частиною цього процесу. Програмне забезпечення перевіряється на відповідність функціо-

[1] IEEE Std 1012-2016 — V&V Software verification and validation

[2] ESA PSS-05-10 1-1 1995 – Guide to software verification and validation

нальних властивостей згідно вимог.

Процес валідації включає в себе перегляд (code review), тестування (модульне, інтеграційне, властивостей), перевірка моделей, аудит, увесь комплекс необхідний для доведення, що продукт відповідає вимогам висунутим при розробці. Такі вимоги формуються на початковому етапі, результатом якого є формальна специфікація.

1.2. Формальна специфікація

Для спрощення процесу верифікації та валідації застосовується математична техніка формалізації постановки задачі — формальна специфікація. Формальна специфікація — це математична модель, створена для опису систем, визначення їх основних властивостей, та інструментарій для перевірки властивостй (формальної верифікаціїї) цих систем, побудованих на основі формальної специфікації.

Існують два фундаментальні підходи до формальних специфікацій: 1) Аглебраїчний підхід, де система описується в термінах операцій, та відношень між ними (або аналітичний метод), та 2) Модельно-орієнотований підхід, де модель створена конструктивними побудовами, як то на базі теорії множин, чи інкаше, а системні операції визначаються тим, як вони змінюють стан системи (конструктивний, або синтетичний метод). Також були створені сімейства послідованих та розподілених мов.

1.2.1. Програмне забезпечення. Найбільш стандартизована та прийнята в обсласті формальної верификації — це нотація Z[3] (Spivey, 1992), приклад модельно-орієнтовоної мови Назавана у честь Ернеста Цермело, роботи якого мали вплив на фундамент математики та аксіоматику теорії множин. Саме теорія множин, та логіка предикатів першого порядку є теорією мови Z. Тут також заслуговують уваги сесійні

[3]ISO/IEC 13568:2002 — Z formal specification notation

типи Кохея Хонди[4], які дозволяють формалізувати протоколи на рівні вбудованої теорії типів.

Інша відома мова формальної специфікації як стандарт для моделювання розподілених систем, таких як телефонні мережі та протоколи, це LOTOS[5] (Bolognesi, Brinksma, 1987), як приклад алгебраїчного підходу. Ця мова побудована на темпоральних логіках, та поведінках залежниих від спостережень. Інші темпоральні мови специфікацій, які можна відзначити тут — це TLA+[6], CSP (Hoare, 1985), CCS[7] (Milner, 1971), Actor Model, BPMN, etc.

[4] `http://mrg.doc.ic.ac.uk/kohei/`

[5] ISO 8807:1989 — LOTOS — A formal description technique based on the temporal ordering of observational behaviour

[6] The TLA+ Language and Tools for Hardware and Software Engineers

[7] J.C.M. Baeten. A Brief History of Process Algebra.

1.2.2. Математичні компоненти. Перші системи комп'ютерної алгебри були розроблені ще під PDP-6 та PDP-10, такі як MATHLAB (MIT), інші ранні системи: MACSYMA (Джоел Мозес), SCRATCHPAD (Ричард Дженкс, IBM), REDUCE (Тони Хирн), SAC-I, пізніше SACLIB (Джорж Коллінз), MUMATH для микропроцессоров (Девід Стоутмаєр) та пізніше DERIVE. Сучасні системи комп'ютерної алгебри: AXIOM послідовних SCRATCHPAD (NAG), MAGMA (Джон Кеннон, Сіднейський університет), MUPAD (Бенно Фуксштейнер, університет міста Падерборн).

1.3. Формальні методи верифікації

1.3.1. Спеціалізовані системи моделювання. Можна виділити три підходи до верифікації. Перший застосовується де вже є певна програма написана на певній мові програмування і потрібно довести ізоморфність цієї програми до доведеної моделі. Ця задача вирішується у побудові теоретичної моделі для певної мови програмування, потім програма на цій мові переводиться у цю теоретичну модель і доводить ізоморфізм цієї програми у побудованій моделі до доведеної моделі. Приклади таких систем та піходів: 1) VST (CompCert, сертифікація C програм); 2) NuPRL (Cornell University, розподілені системи, залежні типи); 3) TLA+ (Microsoft Reseach, Леслі Лампорт); 4) Twelf (для верифікації мов програмування); 5) SystemVerilog (для програмного та апаратного забезпечення).

1.3.2. Мови з залежними типами та індукцією. Другий підхід можна назвати підходом вбудованих мов. Компілятор основової мови перевіряє модель закодовану у ній же. Можливо моделювання логік вищого порядку, лінійних логік, модальних логік, категорний та гомотопічних логік. Процес специфікації та верифікації відбувається в основній мові, а сертифіковані програми автоматично екстрагуються в довільні мови. Приклади таких систем: 1) Coq побудована на мові OCaml від науково-дослідного інституту Франції INRIA; 2) Agda побудовані на мові Haskell від шведського інституту технологій Чалмерс; 3) Lean побудована на мові C++ від Microsoft Research та Універсистету Каргені-Мелона; 4) F* – окремий проект Microsoft Research.

1.3.3. Гомотопічні мови. Приклади таких систем: 1) cubicaltt — [8]CCHM імплементація авторства Андерса Мортберга кубічної теорії типів Саймона Губера; 2) yacctt — ще одна декартова кубічна теорія [9]ABCFHL; 3) Agda –cubical — вбудований кубічний тайпчекер в Агду; 4) Lean — Lean також має вбудований кубічний тайпчекер; 5) RedPRL – кубічна імплементація декартової кубічної теорії ABCFHL.

1.3.4. Системи автоматичного доведення теорем. Третій підхід полягає в синтезі конструктивного доведення для формальної специфікації. Це може бути зроблено за допомогою асистентів доведення теорем, таких як HOL/Isabell, Coq, ACL2, або систем розв'язку задач виконуваності формул в теоріях (Satisfiability Modulo Theories, SMT).

Перші спроби пошуку формального фундаменту для теорії обчислень були покладені Алонзо Черчем та Хаскелем Каррі у 30-х роках 20-го століття. Було запропоноване лямбда числення як апарат який може замінити класичну теорію множин та її аксіоматику, пропонуючи при цьому обчислювальну семантику. Пізніше в 1958, ця мова була втілена у вигляді LISP лауреатом премії тюрінга Джоном МакКарті, який працював в Прінстоні. Ця мова була побудована на конструктивних примітивах, які пізніше виявилися компонентами індуктивних конструкцій та були формалізовані за допомогою теорії категорій Вільяма Лавіра. Окрім LISP, нетипізоване лямбда числення маніфестується у такі

[8]Cyril Cohen, Thierry Coquand, Simon Huber, Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. 2015. `https://5ht.co/cubicaltt.pdf`

[9]Carlo Angiuli, Brunerie, Coquand, Kuen-Bang Hou (Favonia), Robert Harper, Dan Licata. Cartesian Cubical Type Theory. 2017. `https://5ht.co/cctt.pdf`

мови як Erlang, JavaScript, Python. До цих пір нетипізоване лямбла числення є одною з мов у які робиться конвертація доведених программ (екстракція).

Перший математичний прувер AUTOMATH (і його модифікації AUT-68 та AUT-QE), який був написаний для комп'ютерів розроблявся під керівництвом де Брейна, 1967. У цьому прувері був квантор загальності та лямбда функція, таким чином це був перший прувер побудований на засадах ізоморфізма Каррі-Говарда-Ламбека.

ML/LCF або метамова і логіка обчислювальних функцій були наступним кроком до осягнення фундаментальної мови простору, тут вперше з'явилися алебраїчні типи даних у вигляді індуктивних типів, поліноміальних функторів або термінованих (well-founded) дерев. Роберт Мілнер, асистований Морісом та Н'юві розробив Метамову (ML), як інструмент для побудови прувера LCF. LCF був основоположником у родині пруверів HOL88, HOL90, HOL98 та останньої версії на даний час HOL/Isabell. Пізніше були побувані категорні моделі Татсоя Хагіно (CPL, Японія) та Робіна Кокета (Charity, Канада).

У 80-90 роках були створені інші системи автоматичного доведення теорем, такі як Mizar (Трибулєк, 1989). PVS (Оур, Рушбі, Шанкар, 1995), ACL2 на базі Common Lisp (Боєр, Кауфман, Мур, 1996), Otter (МакКюн, 1996).

1.4. Формальні мови та середовища виконання

Усі середовища виконання можно умовно розділити на два класи: 1) інтерпретатори нетипізованого або просто типізованого (рідше з більш потужними системами типів), лямбда числення з можливими JIT оптимізаціями; та 2) безпосередня генерація інструкцій процессора і лінкування цієї програми з середовищем виконання що забезпечує планування ресурсів (в цій області переважно використовується System F типізація).

До першого класу можно віндеси такі віртуалні машини та інтерпретатори як Erlang (BEAM), JavaScript (V8), Java (HotSpot), K (Kx), PHP (HHVM), Python (PyPy), LuaJIT та багато інших інтерпретаторів.

До другого класу можна віднести такі мови програмування: ML, OCaml, Rust, Haskell, Pony. Часто використовується LLVM як спосіб генерації програмного коду, однак на момент публікації статті немає промислового верифікованого LLVM генератора. Rust використовує проміжну мову MIR над LLVM рівнем. Побудова верифікованого компілятора для такого класу систем виходить за межі цього дослідження. Нас тут буде цікавити лише вибір найкращого кандидата для середовижа виконання.

Нійбільш цікаві цільові платформи для виконання программ які побудовані на основі формальних доведень для нас є OCaml (тому, що це основна мова естракту для промислової системи доведення теорем Coq), Rust (тому, що рантайм може бути написаний без використання сміттєзбірника), Erlang (тому, що підтримує неблоковану семантику пі-калкулуса) та Pony (тому, що семантика його пі-калкулуса побудована на імутабельних чергах та CAS-курсорах). У цій роботі ми зосередимося на дослідженні тьох підходів та побудові наступних прототипів.

1.4.1. Формальні інтерпретатори та ОС. Перший прототип, рантайм $_{\text{CPS}}$ – лінивий векторизований інтерпретатор (підтримка SSE/AVX інструкцій) та система управління ресурсами з планувальником лінивих програм та системою черг і CAS курсорсів у якості моделі пі-калкулуса. Розглядається також використання ядра L4 на мові C, верифікованого за допомогою HOL/Isabell, у якості базової операційної системи.

1.4.2. Формальний ввід-вивід. Другий прототип побудований на базі coq.io, що дозволяє використовувати бібліотеки OCaml для промислового програмування в Coq. У цій роботі ми формально показали і продемонстрували коіндуктивний шел та вічно працюючу тотальну програму на Coq. Ця робота проводилася в рамках дослідження системи ефектів для результуючої мови програмування.

1.4.3. Чисті системи типів. Третій прототип – побудова тайпчекера та ектрактора у мову Erlang та O. Ця робота представлена у вигляді PTS тайпчекера OM, який виступає у ролі проміжної мови для повної нормалізації лямбда термів. В роботі використане нерекурсивне кодування індуктивних типів та продемонстрована теж бескінечна тотальна программа у якості способу лінкування з підсистемою вводу-виводу віртуальної машини Erlang. В доповнення до існуючих імплементацій CoC на Haskell (Morte).

1.4.4. Гомотопічні системи типів. Четвертий прототип — імплементація першого кубічного верифікатора на мові Erlang в доповнення до існуючих CCHM (Erlang, Haskell), ABCFHL (Haskell,OCaml).

1.5. Висновки

Як результат цього розділу, було досліджено: i) усі системи доведення теорем; ii) формальні мови програмування; iii) системи верифікації; iv) формальні середовища виконання, та встановлено, що усі такі системи є носіями мовних елементів які згідно теорії типів Мартіна-Льофа можна кластеризувати по наступним мовам, кожна з яких репрезентує правила типу: i) O_λ — нетипизоване λ-числення Чорча; ii) O_π — числення процесів, CCS, CSP або π-числення Мілнера; iii) O_μ — тензорне числення та векторизація (MatLab, Julia, kx, J); iv) O_Π — числення конструкцій (функціональна повнота); v) O_Σ — числення контекстів (контекстуальна повнота); vi) $O_=$ — теорія типів Мартіна-Льофа (логіка); vii) O_W — числення індуктивних конструкцій (матіндукція); viii) O_I — гомотопічна система типів (формальна математика).

РОЗДІЛ 2
КОНЦЕПТУАЛЬНА МОДЕЛЬ СИСТЕМИ ДОВЕДЕННЯ ТЕОРЕМ

Другий розділ описує розвиток концептуальної моделі системи доведення теорем як сукупності: i) середовища виконання, яке складається з інтерпретатора та операційної системи; ii) послідовності формальних мов програмування, кожна наступна з яких, складніша за попередню, має свою операційну семантику, та наслідує усі властивості попередніх мов послідовності.

2.1. Попередні відомості та теорії

Для розповіді про систему доведення теорем, як систему мов програмування будемо використовувати теорію категорії та теорію індуктивних типів для специфікації синтаксисів мов програмування. Перелік необхідних формальних теорій: лямбда числення, теорії індуктивних типів, вищі рівності для гомотопічної системи тощо, містяться в розділі 3. Формальна теорія категорії міститься у розділі 4. Слід зауважити, що ця формалізація проводиться на основі гомотопічної мови програмування побудованої в даному розділі 2.

Визначення 1. (Концепт, Готлоб Фреге). Концепт — це предикат над об'єктом, або іншими словами залежний П-тип з теорії типів Мартіна-Льофа. Об'єкт $x : o$ належить до концепту, тільки якщо сам концепт, параметризований цим об'єктом, населений $p(o) : U$, де $p : \mathrm{concept}(o)$.

Визначення 2. (Система). Визначимо систему як сукупність об'єктів $\mathrm{Ob} : U$ та зв'язків між ними $\mathrm{Hom} : \mathrm{Ob} \to \mathrm{Ob} \to U$.

Визначення 3. (Концептуальна модель). Таким чином концептуальна модель визначається як категорія, об'єкти якої індексовані певною множиною, або залежні від параметра.

Визначення 4. (Синтаксичне дерево). Синтаксичне дерево — це індуктивний тип або дерево Бома, контруктори якого відповідають одному з 5 правил в теорії типів, як правило використовуються три правила: правило формації, інтро-правила та елімінатор.

Визначення 5. (Вище синтаксичне дерево). Синтаксичне дерево в яке додано β та η правила називається вищим синтаксичним деревом.

Визначення 6. (Мова програмування). Мова програмування або мовна категорія — це категорія, об'єкти якої — це maybe-типи сум синтаксичних дерев мов програмування, а морфізми — це стрілки (які містять правила виводу, типизації, нормалізації, екстакції тощо). Приклади синтаксичних дерев: O_{Π}, O_{Σ}, $O_{=}$. Приклади мовних категорій: O_{PTS}, O_{MLTT}, O_{HTS}.

Визначення 7. (Модель). Модель визначимо як систему формальних мов (об'єкти) разом з їх програмами, та мовними перетвореннями (звязки) між ними для яких працює правило асоціативності композиції та правила лівої і правої композиції з одиничними стрілками. Іншими словами будемо розуміти тут категорну модель.

Визначення 8. (Послідовність синтаксичних дерев). Кожна послідовність синтаксичних дерев

$$O_{\Pi} \to O_{\Sigma} \to O_{=} \to O_{W} \to O_{I}. \tag{2.1}$$

генерує відповідну послідовність мов програмування

$$\begin{aligned} O_{PTS}(O_{\Pi}) \to O_{CTX}(O_{\Pi}, O_{\Sigma}) \to O_{EQU}(.., O_{\Sigma}, O_{=}) \to \\ \to O_{ITS}(..., O_{=}, O_{W}) \to O_{HTS}(..., O_{W}, O_{I}). \end{aligned} \tag{2.2}$$

наступним чином. Кожна мова програмування залежить від синтаксису який її визначає та всіх попередніх синтаксисів мов програмування з послідовності. Перша мова програмування містить тільки перший синтаксис. Розкриті сигнатури мають вигляд: $O_{PTS} : O_{\Pi} \to U$, $O_{CTX} : O_{\Pi} \to O_{\Sigma} \to U$, $O_{EQU} : O_{\Pi} \to O_{\Sigma} \to O_{=} \to U$, $O_{ITS} : O_{\Pi} \to O_{\Sigma} \to O_{=} \to O_{W} \to U$, $O_{HTS} : O_{\Pi} \to O_{\Sigma} \to O_{=} \to O_{W} \to O_{I} \to U$.

Таким чином кожна наступна мова програмування містить усі попередні мови програмування, визначені послідовністю синтаксичних дерев,

Визначення 9. (Створення мовної категорії). Мови можна додавати, наприклад $O_{HTS} = O_{\Pi\Sigma=WI}$, для побудови якої необхідно об'єднати у індуктивному типі мови усі індуктивні типи її підмов. Таким чином функтор діє на декартовому добутку синтаксичних дерев мовних категорій та має значення в категорій мовних категорій. Приклад найпотужнішої гомотопічної мови:

$$O_{HTS} = O_{\Pi\Sigma=WI} : O_{\Pi} \to O_{\Sigma} \to O_{=} \to O_{W} \to O_{I} \to U. \tag{2.3}$$

Кожне синтаксичне дерево, як правило, містить конструктори та елімінатори певного одного типу. Але починаючи з O_{ITS} складність типів, які додаються до ядра значно зростає. Таким чином мовні категорії конструюються гранулярно з точністю до включення певного типу в ядро верифікатора.

Визначення 10. (Типи синтаксичних дерев). У розділі 1 були проаналізовані усі мови програмування та середовища виконання, а також спеціалізовані мови моделювання. В результаті чого було встановлено чітки індивідуальні мовні синтаксиси. Кожен синтаксис складається з множини синтаксичних одиниць цієї мови (конструктори індуктивного типу), які відповідають правилам теорії типів Мартіна-Льофа (формації, інтро-правило, елімінатор, β-, та η-правила). Якщо додати β-, та η-правила як рівності у визначення синтаксису, то для представлення потрібні вищі індуктивні типи. Таким чином кожному синтаксичному дереву відповідає певний тип в теорії типів Мартіна-Льофа.

Визначення 11. (Спектральна категорія мов). Так, виділяється наступна послідовність мов, та функторів між ними, де кожна мова-кодомен є складнішою та біль потужною за мову-домен. Система мов є категорією мовних категорій або категорією мов програмування.

$$O_\infty : O_{CPS} \to O_{PTS} \to O_{MLTT} \to O_{ITS} \to O_{HTS} \to \ldots \qquad (2.4)$$

Визначення 12. (Коконтекстуальна категорія мов). Якщо не виділяти певну послідовність мовного ускладнення та розглядати усі суми усієї певної множини мовних синтаксисів, то ми отриміємо коконтек-

Таблиця 2.1

Кластерний аналіз мовних синтаксисів

Синтаксис	Мова програмування або її підмова
O_λ	Нетипизоване λ-числення Чорча
O_π	Числення процесів, CCS, CSP або π-числення Мілнера
O_μ	Тензорне числення та векторизація
O_Π	Числення конструкцій (функціональна повнота)
O_Σ	Числення контекстів (контекстуальна повнота)
$O_=$	Теорія типів Мартіна-Льофа (логіка)
O_W	Числення індуктивних конструкцій (матіндукція)
O_I	Гомотопічна система типів (формальна математика)

стуальну категорію, де об'єкти — це усі можливі мовні категорії побудовані за допомогою усіх перестановок суми мовних синтаксисів, а морфізми це функтори перетворення однієї мовної категорії в іншу мовну категорію. Приклади: $O_{I*} \to O_{\Pi=}$, $O_\Pi \to O_{\Pi\Sigma}$, $O_\Pi \to O_{\Pi\Sigma}$, $O_{\Pi*} \to O_\Pi$.

2.2. Структурне представлення моделі

Виходячи з визначення моделі, вони можуть мати різний набір об'єктів в системі мов програмування. Покажемо приклади ексземплярів які можно породити в рамках цієї моделі.

2.2.1. Мінімальна система. Приклад мінімальної системи, яка містить лише одну мову для доведення теорем та одну мову для виконання програм.

$$\mathsf{PTS}_{\mathsf{CPS}} = \begin{cases} \mathsf{Ob} : \{\mathsf{O}_{\mathsf{CPS}}, \mathsf{O}_{\mathsf{PTS}}\} \\ \mathsf{Hom} : \{1, 2 : \mathbb{1} \to \mathsf{O}_{\mathsf{PTS}}, 3 : \mathsf{O}_{\mathsf{PTS}} \to \mathsf{O}_{\mathsf{CPS}}\} \end{cases} \tag{2.5}$$

Стрілки 1 та 2 визначають модель та базову бібліотеку, а стрілка 3 означає екстракт доведення (якщо таке є) в інтерпретатор. Можна використати графічну мову мереж Петрі для зображення екземпляра моделі системи мов.

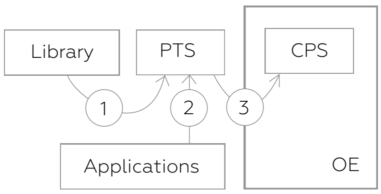

Рис. 2.1. Мінімальна система з чистої мови та інтерпретатора

2.2.2. Максимальна система. Інший приклад системи — це максимальна система, яка містить усі формальні мови програмування та формальне середовище виконання (порядок синтаксичних дерев як параметрів при конструюванні мовної категорії може змінюватися, тут генеалогія HTS не ведеться від MLTT, яке є розгалуженням).

$$\mathrm{Total} = \begin{cases} \mathrm{Ob} : \{O_{\mathrm{CPS}}, O_{\mathrm{PTS}}, O_{\mathrm{MLTT}}, O_{\mathrm{ITS}}, O_{\mathrm{HTS}}\} \\ \mathrm{Hom} : \begin{cases} 1, 2 : \mathbb{1} \to O_{\mathrm{HTS}}, 3 : O_{\mathrm{MLTT}} \to O_{\mathrm{ITS}} \\ 4 : O_{\mathrm{HTS}} \to O_{\mathrm{ITS}}, 5 : O_{\mathrm{ITS}} \to O_{\mathrm{PTS}}, 6 : O_{\mathrm{PTS}} \to O_{\mathrm{CPS}} \end{cases} \end{cases} \tag{2.6}$$

За допомогою мереж Петрі це можна відобразити наступним чином:

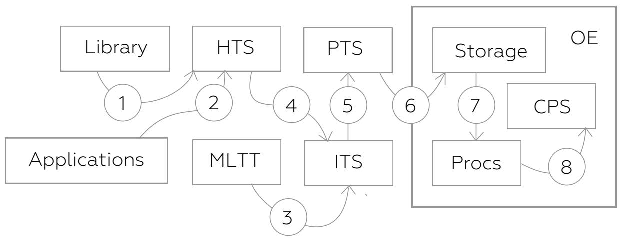

Рис. 2.2. Кубічна та чиста системи типів та середовище виконання

2.3. Формальні мови програмування

Тут йдеться про мови програмування придатні для доведення теорем, та їх таксономію від найелементарніших (чистої системи з одним типом П) до найпотужніших гомотопічних систем. Одна така гомотопічна система є кінцевим завданням цього розділу — побудова моделі гомотопічного верифікатора. В процесі його побудови в цьому розділі ми розглянемо під мікроскопом складові частини його нижчих мовних рівнів.

Застосуємо категорну семантику для мов програмування і будемо розглядати мови програмування як моноїдальні мовні категорі, об'єкти яких є просторами усіх програм цих мов програмування, а морфізми — правила верифікації та компіляції цих мов. Морфізми між мовними категоріями в категорї мов програмування — це функтори підвищення та пониження складності мови, подібно до того як діють морфізми в контекстуальних категоріях. Морфізм деконструює або конструює за допомогою Either-типу або Σ-типу індуктивний тип мови програмування.

Мови розкладаються у спектральну (індексовану натуральними числами $\mathbb{N} \rightarrow \mathbb{U}$) послідовність мов, кожен елемент якої є мовою програмування, яка не містить синтаксичне дерево вищої мови програмування.

2.3.1. Чиста система типів $\mathrm{O}_{\mathrm{PTS}}$**.** Чиста ситема або числення конструкцій або система з одим типом або система з однією аксіомою, продовжує традиції елементарних пруверів в стилі першого AUTOMATH та сучасних Henk, Morte, Cedile, Om.

Визначення 13. (Мовна категорія чистої мови $\mathrm{O}_{\mathrm{PTS}}$).

$$\mathrm{O}_{\mathrm{PTS}} = \begin{cases} Ob : \{\, X : maybe\ PTS, target : maybe\ CPS \,\} \\ Hom : \begin{cases} type, norm : X \rightarrow X, extract : X \rightarrow target \\ certify : X \rightarrow target = type \circ norm \circ extract \end{cases} \end{cases} \tag{2.7}$$

Визначення 14. (Синтаксис мовної категорії $\mathrm{O}_{\mathrm{PTS}}$). Чиста мова $\mathrm{O}_{\mathrm{PTS}}$ містить лише синтаксис одного типу, П-типу. Така теорія називається теорією з одним типом, або з однією аксіомою.

```
data PTS = ppure (_: pts PTS)
```

Вона описана в літературі як Calculus of Construction (Кокан), Pure Type System (Стемп, Фу).

Визначення 15. (Синтаксичне дерево O_{Π}).

```
data pts (lang: U)
   = star (n: nat)
   | var (x: name) (l: nat)
   | pi (x: name) (l: nat) (f: lang)
   | lambda  (x: name) (l: nat) (f: lang)
   | app (f a: lang)
```

2.3.2. Теорія типів Мартіна-Льофа O_{MLTT}. Мова теорії типів є сучасною основою всіх пруверів з залежними типами, такими, наприклад, як NuPRL та Agda. Багато так званих $\Pi\Sigma$ пруверів імплементують MLTT серед таких як: $\Pi\Sigma$[1], $\Pi\forall$[2].

Визначення 16. (Мовна категорія O_{MLTT}).

$$O_{MLTT} = \begin{cases} Ob : \{\, maybe\ MLTT \,\} \\ Hom : \begin{cases} type, norm : Ob \to Ob \\ certify : Ob \to Ob = type \circ norm \end{cases} \end{cases} \tag{2.8}$$

Визначення 17. (Синтаксис мовної категорії O_{MLTT}). Мова O_{MLTT} включає в себе синтаксиси трьох типів теорії Мартіна-Льофа: O_{Π}, O_{Σ}, $O_{=}$.

[1] https://github.com/zlizta/pisigma-0-2-2

[2] https://github.com/sweirich/pi-forall

```
data MLTT = mpure (_: pts MLTT)
          | msigma (_: exists MLTT)
          | mid (_: identity MLTT)
```

Визначення 18. (Синтаксичне дерево O_{Σ}). Також можна до чистої системи додати Σ-тип, піднявши типову систему до мови $O_{MLTT-72}$ або $O_{\Pi\Sigma}$:

```
data exists (lang: U)
   = sigma (n: name) (a b: lang)
   | pair (a b: lang)
   | fst (p: lang)
   | snd (p: lang)
```

Визначення 19. (Синтаксичне дерево $O_{=}$). Додавши тип рівності можно підняти систему ще на одну сходинку, до $O_{MLTT-84}$ або $O_{\Pi\Sigma=}$:

```
data identity (lang: U)
   = id (t a b: lang)
   | id_intro (a b: lang)
   | id_elim (a b c d e: lang)
```

2.3.3. Система індуктивних типів O_{ITS}.

Визначення 20. (Мовна категорія O_{ITS}).

$$O_{ITS} = \begin{cases} Ob : \{\, X : maybe\ ITS, target : maybe\ CPS \,\} \\ Hom : \begin{cases} type, norm, induction : X \rightarrow X, extract : X \rightarrow target \\ certify : X \rightarrow target \\ cerfity = type \circ norm \circ induction \circ extract \end{cases} \end{cases} \tag{2.9}$$

Мова індуктивних типів дозволяє безпосередньо кодувати індуктивні типи, не використовуючи схеми кодування Бома, містить усі попередні мовні синтаксиси: $O_=$, O_Σ, O_Π.

Визначення 21. (Синтаксичне дерево мовної категорії O_{ITS}).

```
data ITS = ipure (_: pts ITS)
         | isigma (_: exists ITS)
         | iid (_: identity ITS)
         | iITS (_: ind ITS)
```

Мова містить наступні допоміжні визначення: і) телескопу, який містить послідовність елементів мови; іі) розгалуження, як конструкцій case оператора; ііі) імен конструкторів індуктивного типу.

```
data tele   (A: U) = emp | tel (n: name) (b: A) (t: tele A)
data branch (A: U) =        br (n: name) (args: list name) (term: A)
data label  (A: U) =       lab (n: name) (t: tele A)
                         | com (n: name) (t: tele A) (dim: list name)
                               (s: list  (prod (prod name bool) A))
```

Визначення 22. (Синтаксичне дерево O_*). Правило формації, конструктора та елімінатора визначається синтаксичним деревом O_*:

```
data ind (lang: U)
   = datum  (n: name) (t: tele lang) (labels:   list (label lang))
   | case   (n: name) (t: lang)      (branches: list (branch lang))
   | ctor   (n: name)                (args:     list lang)
```

2.3.4. Гомотопічна система типів O_{HTS}.

Визначення 23. (Мовна категорія O_{HTS}).

$$O_{\mathrm{HTS}} = \begin{cases} Ob : \{\, maybe\ HTS \,\} \\ Hom : \begin{cases} type, norm : Ob \to Ob \\ certify : Ob \to Ob = type \circ norm \end{cases} \end{cases}$$

Визначення 24. (Синтаксис мовної категорії O_{HTS}). Синтаксис гомотопічної мовної категорії містить усі попередні мовні синтаксиси: O_{I}, O_W, $O_=$, O_Σ, O_Π:

```
data HTS = hpure (_: pts HTS)
         | hsigma (_: exists HTS)
         | hid (_: identity HTS)
         | hind (_: ind HTS)
         | homotopy (_: hts HTS)
```

Гомотопічна типа наслідує O_{ITS} але модифіковану з Path-типом в індуктивних визначеннях, структурою композиції, анонсує Path-тип (формація, конструктор, та елімінатор) як лямбда функцію на відрізку, а також склейку типів у всесвіті та склейку змінних з відповідними елімінаторами.

Визначення 25. (Синтаксичне дерево O_{I}).

```
data hts (lang: U)
   = path (t a b: lang)
   | plam (n: name) (a: alg) (b: lang)
   | papp (f: name) (a: lang) (p: alg)
   | comp_ (a b: lang)
   | fill_ (a b c: lang)
   | glue_ (a b c: lang)
   | glue_elem (a b: lang)
   | unglue_elem (a b: lang)
```

Таким чином, $\mathrm{O}_{\mathrm{HTS}}$ містить два Id-типа, один унаслідований від $\mathrm{O}_{=}$, а інший який міститься в синтаксичному дереві O_{I}.

2.4. Формальне середовище виконання

Формальне середовище виконання складається з інтерпретатора (нетитизованого λ-числення) та числення акторів (процесів, черг, таймерів). Інтерпретатор та операційна система включені в систему доведення теорем для уніфікації всіх сигнатур системи та формалізації самого інтерпретатора як системи виконання. Слід зазначити, що не завжди є змога зробити екстракт в $\mathrm{O}_{\mathrm{CPS}}$, тому об'єкти мовних категорій є $\mathfrak{maybe}$-типами.

$$\mathrm{O}_{\mathrm{CPS}} : \mathrm{O}_{\lambda} \to \mathrm{O}_{\pi} \to \mathrm{O}_{\mu} \to \mathrm{U}$$

Далі буде йтися тільки про формальні інтерпретатори, так як вони є найбільш компакними формами мов для верифікації (в порівнянні з моделями System F). Таким чином будемо розглядати формальне середовище виконання, як сукупність інтерпретатора та операційної системи.

В цьому розділі ми побудуємо надшвидку імплементацію інтерпретатора, яка цілком, разом зі своїми програмами, розміщується в кешпамяті першого рівня процесора, та здатна до AVX векторизацій засобами мови Rust. Як промислова опція, підтримується також екстракт в байт-код інтерпретатора BEAM віртуальної машини Erlang.

2.4.1. Категорія середовища виконання $\mathrm{O}_{\mathrm{CPS}}$.

Визначення 26. (Категорія середовища виконнанняO_{PS}).

$$O_{CPS} = \begin{cases} \mathsf{Ob} : \{\ \mathit{maybe}\ \mathsf{CPS}\ \} \\ \mathsf{Hom} : \{\ \mathit{eval} : \mathsf{Ob} \to \mathsf{Ob}\ \} \end{cases}$$

Синтаксис середовища виконання може містити наступні синтаксиси: O_λ, O_π, O_μ.

Визначення 27. (Синтаксис мовної категорії O_{HTS}).

```
data CPS = church (_: lambda CPS)
         | milner (_: pi     CPS)
         | tensor (_: mu     CPS)
```

Визначення 28. (Синтаксичне дерево O_λ). Інтерпретатор визначається своїм трьома конструкторами: номер змінної (індекс де Брейна), лямбда функція та її апплікація:

```
data lambda = var (x: nat)
            | lam (l: nat) (d: cps)
            | app (f a: cps)
```

Мовою інтерпретаторів є нетипизоване лямбда числення, однак в залежності від складності інтерпретатора це дерево може виглядати порізному.

Визначення 29. (Синтаксичне дерево O_π). Правило формації, конструктора та елімінатора визначається синтаксичним деревом O_*:

```
data pi (lang: U)
   = process (protocol: lang)
   | spawn (cursors: lang) (core: nat) (program: lang)
   | snd (cursor: lang) (data: lang)
   | rcv (cursor: lang)
```

```
| pub (size: nat)
| sub (cursor: lang)
```

Кожна секція цієї глави буде присвячена цим мовним компонентам системи доведення теорем. В кінці розділу дається повна система, яка включає в себе усі мови та усі мовні перетворення.

2.5. Чиста система типів PTS^{∞}

IEEE[3] стандарт та регуляторні документи ESA[4] визначають інтрументи та підходи до виробничого процесу верифікації та валідації. Найбільш розвинені та потужні засоби вимагають застосування математичних мов та нотацій. Ера верифікованої математики була започаткована верифікатором AUTOMATH [1] (де Брейн) розробленого під керівництвом де Брейна, а також розвиток теорії типів Мартіна-Льофа [2]. Сьогодні ми маємо Lean, Coq, F*, Agda мови які використовують числення конструкцій, Calculus of Constructions [3] (CoC) та числення індуктивних типів (Calculus of Inductive Constructions [4] (CiC). Пізніше учень де Брейна, Хенк Барендрехт класифікував послаблені чисті системи типів по трьом осям та візуазізував це за допомогою лямбда-куба [5]. Чисті мови програмування вже були імплементовані раніше (Morte[5] Габрієля Гонзалеза, Henk [6] Еріка Мейера). Чисті системи типів це системи з однм Π-типом (або ще і Σ як в ECC [7], Ope), з

[3]IEEE Std 1012-2016 — V&V Software verification and validation

[4]ESA PSS-05-10 1-1 1995 – Guide to software verification and validation

[5]Gabriel Gonzalez. Haskell Morte Library https://github.com/Gabriel439/Haskell-Morte-Library

можливими розширеннями, такими як PTS^{∞} з бескінечною кількістю всесвітів [8] (Сохацький), Cedile з self-типами [9] [10] (Стамп, Фу), система з K-правилами [11] (Барте).

Головна мотивація чистих систем – це простота аналізу ядра верифікатора, можливість застосування сильної нормалізації та довірена зовнішня верифікація та сертифікація завдяки простоті верифікатора (type checker), це означає, що алгоритм верифікації повинен бути настільки простим, аби можна було просто імплементувати його на будь-якій мові програмування. Приклади застовування тут можуть бути: 1) формальна мова блокчейн контрактів (Pluto [6]); 2) сертифіковані обчислення для інтерпретаторів; 3) платіжні системи.

2.5.1. Генерація сертифікованих програм. Згідно ізоморфізму Каррі-Говарда-Ламбека або інтерпретації Брауера-Гейтінга-Колмогорова існує взаємноознозначна відповідність між доведеннями теорем (або пруфтермами) та лямбда функціями в теорії типів Мартіна-Льофа [2]. Так як специфікація та доведення її відоповідності для певної програми відбувається за допомогою мови з залежними типами, ми можемо екстрагувати цільову імплементацію (зі стертою інформацією про типи) сертифікованої программи в довільну мову програмування. У якості такої цільової мови підходять майже усі інтерпретатор безтипового лямбда числення, такі як JavaScript, Erlang, PyPy, LuaJIT, K.

Більш розвинені практики та підходи до кодогенерації та екстрагу-

[6]Rebecca Valentine. Formal Specification of the Plutus Core Language. 2017. https://iohk.io/research/papers/#JT5XKNBP

ванню сертифікованих програм полягає у генерації C++ чи Rust програм, або програм для нижчих систем лямюда-кубу, таких як System F або System F_ω. У цій роботі представлений екстракт в мову Erlang у якості цільового інтерпретатору.

Таблиця 2.2

List of languages, tried as verification targets

Target	**Class**	**Intermediate**	**Theory**
JVM	interpreter/native	Java	F-sub[7]
JVM	interpreter/native	Scala	System F-omega
CLR	interpreter/native	F#	System F-omega
GHC	compiler/native	Haskell	System D
GHC	compiler/native	Morte	CoC
GHC,OCaml	compiler/native	Coq	CiC
O,BEAM	interpreter	Om	PTS^∞
JavaScript	interpreter/native	PureScript	System F

PTS синтаксиси. Мінімальне ядро з однією аксіомою сприймає декілька лямбда ситаксисів. Перший синтаксис сумісний з системою програмування morte[8], та походить від неї. Інший синтаксис сумісний з синтаксисом cubical[9]. Планувалося також підтримати синтаксис caramel[10].

[8]http://github.com/Gabriel439/Haskell-Morte-Library

[9]http://github.com/mortberg/cubicaltt

[10]https://github.com/MaiaVictor/caramel

$$\begin{cases} \mathsf{Sorts} = \mathsf{U}.\{\mathsf{i}\},\ \mathsf{i} : \mathsf{Nat} \\ \mathsf{Axioms} = \mathsf{U}.\{\mathsf{i}\} : \mathsf{U}.\{\mathsf{inc\ i}\} \\ \mathsf{Rules} = \mathsf{U}.\{\mathsf{i}\} \rightsquigarrow \mathsf{U}.\{\mathsf{j}\} : \mathsf{U}.\{\mathsf{max\ i\ j}\} \end{cases} \tag{2.10}$$

Мова програмування Ом – це мова з залежними типами, яка є розширенням числення конструкцій (Calculus of Constructions, CoC) Тері Кокана. Саме з числення конструкцій починається сучасна обчислювальна математика. В додаток до CoC, наша мова Ом має предикативну ієрархію індексованих всесвітів. В цій мові немає аксіоми рекурсії для безпосереднього визначення рекурсивних типів. Однак в цій мові вцілому, рекурсивні дерева та корекурсія може бути визначена, або як кажуть, закодована. Така система аксіом називається системою з однією аксіомою (або чистою системою), тому що в ній існує тільки Пі-тип, а для кожного типу в теорії типів Мартіна Льофа існує п'ять конструкцій: формація, інтро, елімінатор, бета та ета правила.

Усі терми підчиняються системі аксіом **Axioms** всередині послідовності всесвітів **Sorts** та складність залежного терму відповідає максимальній складності домена та кодомена (правила **Rules**). Таким чином визначається простір всесвітів, та його конфігурація може бути записана згідно нотації Барендрехта для систем з чистими типами:

Проміжна мова чистої системи типів Ом базується на мові Henk [6], вперше описаній Еріком Мейером та Саймоном Пейтоном Джонсом в 1997 році. Пізніше Габріель Гонзалез імплементував на мові Haskell верифікатор з посиланням на Henk, та використував кодування Бома для нерекурсивного кодування рекурсивних індуктивних типів. Ця мова базується лише на Π-типі, λ-функції, її елімінатора аплікації, β-редукції та η-експансії. Дизайн мови Ом нагадує дизайн мов Henk та Morte. Ця мова призначена бути максимально простою (повна імплементація займає 300 рядків), формально верифікованою, здатною продукувати сертифіковані програми та розповсюджувати їх за межі комп'ютера по мережах та недовірених каналах зв'язку, та компілювати (верифікувати та екстрагувати) на цільових платформах за допомогою тієї ж мови Ом, можливо імплементовоної на іншій мові програмування та вбудованої в основну систему.

2.5.2. Синтаксис. Синтаксис PTS сумісний з численням конструкцій (CoC) Тері Кокана, та такими мовами як Morte та Henk. Однак в системі PTS присутній індекс для всесвітів який представлений натуральними числами. Тут наведений синтаксис у BNF нотації

```
I := #list #nat
U := * + * . #nat
O := U + I + ( O ) + O O + O → O
   + λ ( I : O ) → O
   + ∀ ( I : O ) → O
```

Тут + — сумма виразів, '.' — конкатенація терміналів без пробілу, := — оператор визначення BNF-правила, #empty, #nat, #list — вбудовані типи BNF-нотації — синтаксичні елементи BNF нотації, а *,:, →, (,), λ, ∀ – термінали або синтаксичні елементи мови програмування. Еквівалентне визначення як ініціальний об'єкт категорій O_{PTS} або O_{Π} який може вмістити цей синтаксис містить всі правила виводу внутрішньої мови категорії.

```
data pts (lang: U)
   = star               (n: nat)
   | var    (x: name) (l: nat)
   | pi     (x: name) (l: nat) (d c: lang)
   | remote (n: name) (n: nat)
   | lambda (x: name) (l: nat) (d c: lang)
   | app                       (f a: lang)
```

2.5.3. Всесвіти. Мова PTS^{∞} – це лямбда числення з залежними типами вищого порядку, розширення числення конструкцій Кокана, або системи P_{ω} Барендрехта, з предикативною (імпредикативною) ієрархією індексованих всесвітів. Це розширення мотивоване консистентністю [12] в залежній теорії типів та неможливістю кодування парадоксів Жирара-Хуркенса-Рассела[11]. Також для забезпечення консистентності в мові PTS відсутня аксіома `Fixpoint`, хоча за допомогою рекурсивного трактування конструктора `remote`, така можливість зберігається.

$$U_0 : U_1 : U_2 : U_3 : \ldots$$

Де U_0 — імпредикативний всесвіт, U_1 — перший предикативний всесвіт, U_2 — другий предикативний всесвіт, U_3 — третій предикативний всесвіт і т.д.

$$\frac{o : Nat}{U_o} \qquad (S)$$

[11]Так званий парадокс голяра який виникає в системах $U : U$

Предикативні всесвіти. Всі терми підпорядковуються системі аксіом A для послідовності всесвітів S. Складність R залежності термів дорівнює максимальнії складної термів з яких складаєтсья формула (або вираз мови). Система всесвітів описується згідно SAR-нотації Барендрехта. Зауважте, що предикативні всесвіти несумісні в Бом кодуванням, але ви можете переключати предикативність.

$$\frac{i : Nat, j : Nat, i < j}{U_i : U_j} \tag{A_1}$$

$$\frac{i : Nat, j : Nat}{U_i \to U_j : U_{max(i,j)}} \tag{R_1}$$

Імпредикативні всесвіти. Стягуваний імпредикативний простір внизу ієрархії є єдиним можливим розширенням предикативної ієрархії для того аби вона залишалась консистентною. Однак в чистій системі типів PTS підтримується ієрархія бескінечних імпредикативних всесвітів.

$$\frac{i : Nat}{U_i : U_{i+1}} \tag{A_2}$$

$$\frac{i : Nat, \quad j : Nat}{U_i \to U_j : U_j} \tag{R_2}$$

2.5.4. Контексти. Контексти моделюються словником з іменами змінних в верифікаторі. Він може бути типизований як `list Sigma`. Правило елімінації тут не дається, після використання функції верифікації, словник вивільняється з пам’яті.

$$\frac{}{\Gamma : \mathsf{Ctx}} \qquad \text{(Ctx-formation)}$$

$$\frac{\Gamma : \mathsf{Ctx}}{\varnothing : \Gamma} \qquad \text{(Ctx-intro}_1\text{)}$$

$$\frac{A : U_i, \quad x : A, \quad \Gamma : \mathsf{Ctx}}{(x : A)\ \vdash\ \Gamma : \mathsf{Ctx}} \qquad \text{(Ctx-intro}_2\text{)}$$

2.5.5. Операційна семантика. Операційна семантика — це правила обчислення, або β-,η-правила фьюжену інтро-правила та елімінаторів. для визначення яких необхідно визначити: 1) інтро-правила, їх тип (правило формації), та класс (тип правила формації); 2) правило елімінації та залежної елімінації (індукції). Таким чином будемо вважати, що операційна семантика системи типів O_{PTS} буде складатися з 5 правил: формації, інтро-правило, залежний елімінатор (індукція), β-редукція або правило обчислення, η-експансія або правило унікальності.

$$\frac{A : U_i\ ,\ x : A \vdash B : U_j}{\Pi\ (x : A) \to B : U_{p(i,j)}} \qquad (\Pi\text{-formation})$$

$$\frac{x : A \vdash b : B}{\lambda\ (x : A) \to b : \Pi\ (x : A) \to B} \qquad (\lambda\text{-intro})$$

$$\frac{f : (\Pi\ (x : A) \to B) \quad a : A}{f\ a : B\ [a/x]} \qquad \text{(App-elimination)}$$

$$\frac{x : A \vdash b : B \quad a : A}{(\lambda\,(x : A) \to b)\; a = b\,[a/x] : B\,[a/x]} \qquad (\beta\text{-computation})$$

$$\frac{\pi_1 : A \quad u : A \vdash \pi_2 : B}{[\pi_1/u]\,\pi_2 : B} \qquad (\text{subst})$$

Перелік теорем (специфікації) для чистої системи типів можуть бути прямо вбудовані в теорію типів, таким чином ми отримуємо логічний фреймворк для перевірки імплементації залежної теорії.

```
PTS (A: U): U
  = (Pi_Former: (A -> U) -> U)
  * (Pi_Intro: (B: A -> U) -> ((a: A) -> B a) -> (Pi A B))
  * (Pi_Elim: (B: A -> U) (a: A) -> (Pi A B) -> B a)
  * (Pi_Comp1: (B: A -> U) (a : A) (f: Pi A B) ->
    Equ (B a) (Pi_Elim B a (Pi_Intro B f)) (f a))
  * ((B: A -> U) (a: A) (f: Pi A B) ->
    Equ (Pi A B) f (\(x:A) -> f x))
```

Доведення цих теорем дано в модулі базової бібліотеки розділу 3. Також можна повитися на інші доведення [5]. Рівняння обсислювальної семантики (бета та ета правила) визначаються за допомогою Path-типів, які визначаються $O_=$ або O_I мовним синтаксисом.

Ці рівняння обчислювальної семантики представлені тут як Path-тип в вищій мові. В чистій системі типів PTS з бескінечною кількістю всесвітів ми додається в AST remote конструктор для завантаження файлів з локального довіреного сховища. Рекурсія по цьому конструктору заборонена.

Індекси де брейна діють локально в межах одного імені. При доданні існуючого імені в контекст збільшується індекс цього імені.

Таким чином PTS верифікатор чистої системи типів відрізняється від канонічного приклада алгоритма верифікації CoC [3]. Він включає наступні функції мовної категорії: **підстановка**, **зсув імені**, **нормалізація** термів, **рівність** за визначенням та **верифікація**.

2.5.6. Перевірка типів. Для перевірки типів застосовується наступний алгоритм верифікації, який є основаю усіх залежних систем. В чистих системах потрібно бути обережним з `remote` конструктором. Він використовуються для завантаження типів з локального довіреного сховища. При дозволі рекурсії по `remote` конструктору можливо реалізувати self-типи [10] [9].

```
type (:star,N)       D → (:star,N+1)
     (:var,N,I)      D → :true = proplists:is_defined N B, om:keyget N D I
     (:remote,N)     D → om:cache (type N D)
     (:pi,N,0,I,O)   D → (:star,h(star(type I D)),star(type O [(N,norm I)|D]))
     (:fn,N,0,I,O)   D → let star (type I D), NI = norm I
                          in (:pi,N,0,NI,type(O,[(N,NI)|D]))
     (:app,F,A)      D → let T = type(F,D),
                             (:pi,N,0,I,O) = T, :true = eq I (type A D)
                          in norm (subst O N A)
```

2.5.7. Індекси де Брейна. Зсув переіменовує змінну N в контексті P, тобто додає одиницю для лічильника цієї змінної.

```
sh (:star,X)       N P → (:star,X)
   (:var,N,I)      N P → (:var,N,I+1) when I >= P
                       → (:var,N,I)
   (:remote,X)     N P → (:remote,X)
   (:pi,N,0,I,O)   N P → (:pi,N,0,sh I N P,sh O N P+1)
   (:fn,N,0,I,O)   N P → (:fn,N,0,sh I N P,sh O N P+1)
```

```
          (:app,L,R)     N P → (:app,L,R)
```

2.5.8. Підстановка, нормалізація, рівність. Підстановка заміняє змінну у виразі на певний терм.

```
 sub (:star,X)      N V L → (:star,X)
     (:var,N,L)     N V L → V
     (:var,N,I)     N V L → (:var,N,I-1) when I > L
     (:remote,X)    N V L → (:remote,X)
     (:pi,N,0,I,O)  N V L → (:pi,N,0,sub I N V L,sub O N (sh V N 0) L+1)
     (:pi,F,X,I,O)  N V L → (:pi,F,X,sub I N V L,sub O N (sh V F 0) L)
     (:fn,N,0,I,O)  N V L → (:fn,N,0,sub I N V L,sub O N (sh V N 0) L+1)
     (:fn,F,X,I,O)  N V L → (:fn,F,X,sub I N V L,sub O N (sh V F 0) L)
     (:app,F,A)     N V L → (:app, sub F N V L,sub A N V L)
```

Нормалізація виконує підстановку при аплікаціях до функцій (бета-редукція) за допомогою рекурсивного спуску по конструкторам синтаксичного дерева.

```
norm (:star,X)      → (:star,X)
     (:var,X)       → (:var,X)
     (:remote,N)    → cache (norm N [])
     (:pi,N,0,I,O)  → (:pi,N,0,norm I,norm O)
     (:fn,N,0,I,O)  → (:fn,N,0,norm I,norm O)
     (:app,F,A)     → case norm F of
                          (:fn,N,0,I,O) → norm (subst O N A)
                                     NF → (:app,NF,norm A) end
```

Рівність за визначенням перевіряє рівність Erlang термів.

```
  eq (:star,N)         (:star,N)         → true
     (:var,N,I)        (:var,(N,I))      → true
     (:remote,N)       (:remote,N)       → true
     (:pi,N1,0,I1,O1)  (:pi,N2,0,I2,O2)  →
          let :true = eq I1 I2
```

```
      in eq O1 (subst (shift O2 N1 0) N2 (:var,N1,0) 0)
 (:fn,N1,0,I1,O1) (:fn,N2,0,I2,O2) →
     let :true = eq I1 I2
      in eq O1 (subst (shift O2 N1 0) N2 (:var,N1,0) 0)
 (:app,F1,A1)       (:app,F2,A2)   → let :true = eq F1 F2 in eq A1 A2
 (A,B)                             → (:error,(:eq,A,B))
```

2.5.9. Використання мови. Тут буде показано використання мови PTS.

```
$ ./om help me
[{a,[expr],"to parse. Returns {_,_} or {error,_}."},
 {type,[term],"typechecks and returns type."},
 {erase,[term],"to untyped term. Returns {_,_}."},
 {norm,[term],"normalize term. Returns term's normal form."},
 {file,[name],"load file as binary."},
 {str,[binary],"lexical tokenizer."},
 {parse,[tokens],"parse given tokens into {_,_} term."},
 {fst,[{x,y}],"returns first element of a pair."},
 {snd,[{x,y}],"returns second element of a pair."},
 {debug,[bool],"enable/disable debug output."},
 {mode,[name],"select metaverse folder."},
 {modes,[],"list all metaverses."}]

$ ./om print fst erase norm a "#List/Cons"
   \ Head
-> \ Tail
-> \ Cons
-> \ Nil
-> Cons Head (Tail Cons Nil)
ok
```

2.5.10. Обмеження. Обмеження: 1) неможливість визначити рекурсію та індукцію без fixpoint аксіоми; 2) кодування Бома повинно бути позитивно-рекурсивним; 3) неможливість побудови великого елімінатора, вивести тип з даних; 4) неефективність сім'ї лямбда кодувань вцілому (Парігот, Скотт, Бом).

2.5.11. Екстракти. Мова Ом передбачає автоматичну генерацію сертифікованих програм в цільові платформи. Сертифікая полягаю у візуальному доведенню однієї стрілки ізоморфізма λ-функції в залежній теорії типів та λ-фунції в нетипизованому лямбда численні.

```
ext  (:var,X,N,F)       → (:var,X)
     (:app,A,B,N,F)     → (:call,N,ext(F,A,N),[ext(F,B,N)])
     (:fn,S,_,I,O,N,F)  → (:fun,N,(:clauses,[{:clause,N,
                                   [(:var,N,S)],[],[ext(F,O,N)]}]))
                     _  → []
```

Так працьє функція екстракту в Erlang з системи типів PTS$^\infty$. Erlang-версія Ом повинна бути зручна для використання для віртуальних машин LING та BEAM. Оскільки цей екстракт генерує AST дерево Erlang (подідбно до Elixir), результуючий код подається повністю на весь стек оптимізаційного компілятора Erlang включаючи Erlang Core, тому весь модуль екстракта займає 30 рядків.

2.5.11.1. Інтерпретатори. З практичною точки зору, мова Ом є способом використовувати залежні типи та специфікації побудовані за їх допомогу на мові Erlang. Завдяки глибокій інтеграції з Erlang вдалося мімізувати імплементацію системи до 300 рядків. Екстракт в інтерпретатор O_{PTS} (чи інші) є альтернативною опцією для Ом. Також мова Ом може бути легко портована на інші мови.

2.5.11.2. LLVM. Більш складна опція генерації сертифікованих програм — це генерація машинного коду, з використанням або без використання допоміжних проміжних мов таких як LLVM та MIR. Тому що для цього потрібно верифікувати модель асемблера та процесора а також його оптимізатора, так як зі складністю синтаксичного дерева росте складність та величина терму-доведення будь-яких властивостей.

2.5.11.3. FPGA. Інша, не мен складна, або ще більш складна опція є безпосередня генерація VHDL моделей (наприклад, clash).

2.6. Система індуктивних типів ITS

Індуктивні синтаксиси та кодування можуть підтримуватися за допомогою системи модулів. Кожна система модулів може самостійно (у вигляді ефектів), або за допомогою лямбда кодувань попередньої мови PTS рівня, зберігати та оперувати індуктивними типами даних.

```
def := data id tele = sum + id tele : exp = exp +
       id tele : exp where def
exp := cotele*exp + cotele → exp + exp → exp + (exp) + app + id +
       (exp,exp) + \ cotele → exp + split cobrs + exp .1 + exp .2

  0 := #empty          imp    := [ import id ]
brs := 0 + cobrs       tele   := 0 + cotele
app := exp exp         cotele := ( exp : exp ) tele
 id := [ #nat ]        sum    := 0 + id tele + id tele | sum
```

```
ids := [ id ]          br     := ids → exp
cod := def dec         mod    := module id where imp def
dec := 0 + codec       cobrs  := | br brs
```

Індуктивні синтаксиси будуються на телескопах Диб'єра, конструкторах сум, та їх елімінаторах.

```
data tele   (A: U) = emp | tel (n: name) (b: A) (t: tele A)
data branch (A: U) =        br (n: name) (args: list name) (term: A)
data label  (A: U) =       lab (n: name) (t: tele A)
                         | com (n: name) (t: tele A) (dim: list name)
                               (s: list (prod (prod name bool) A))

data ind (lang: U)
   = datum  (n: name) (t: tele lang) (labels:   list (label lang))
   | case   (n: name) (t: lang)      (branches: list (branch lang))
   | ctor   (n: name)                (args:     list lang)
```

2.6.2. Поліноміальні функтори. Існує два види формальної рекурсії: 1) перша з найменшою нерухомою точкою (як $F_A(X) = 1 + A \times X$ або $F_A(X) = A + X \times X$), іншими словами рекурсія з базою (термінується 1 або A). Списки та дерева є прикладами таких рекурсивних структур з nil та leaf термінальними конструкторами (або рекурсивні суми). 2) друга з найбільшою нерухомою точкою, або рекурсія без бази (як $F_A(X) = A \times X$) — така рекурсія не термінована на рівні типів, та моделює нетерміновані послідовності, процеси тощо (або рекурсивні добутки). Кодування найменшою нерухомою точкою ще називається кодуванням добре-визначиними деревами або кодування поліноміальними функторами.

Натуральні числа: $\mu\, X \to 1 + X$

Списки елементів A: $\mu\, X \to 1 + A \times X$

Лямбда числення: $\mu\, X \to 1 + X \times X + X$

Потоки: $\nu\, X \to A \times X$

Потенційно нескінченний список елементів A: $\nu\, X \to 1 + A \times X$

Кінцеве дерево: $\mu\, X \to \mu\, Y \to 1 + X \times Y = \mu\, X = \mathsf{List}\ X$

Для цих кодувань існує аналог кодування Чорча, який розповсюджує кодування чистими функціями з нетипизованого лямбда числення до П-типу. Таке кодування називається кодуванням Бома-Беррардуччі, а просто кодування Бома. Воно дозволяє кодувати індуктивні типи даних П-типами чистими функціями. Проте як було показано Жеверсом [13] неможливо побудувати принцип індукції в чистих системх без використання в явному чи прихованому вигляді `Fixpoint` аксіоми. Також неможливо побудувати J елімінатор Id типу закодованого в Бом кодуванні, а також елімінатори гомотопічних примітивів, наприклад елімінатори гомотопічного візрізка як `funExt`, `homotopy`.

2.6.3. Кодування Бома. Тип даних List над даним типом A, може бути представлений як ініціальні алгебра $(\mu L_A, in)$ функтору $L_A(X) = 1 + (A \times X)$. Позначається $\mu L_A = \mathsf{List}(A)$. Функції-конструктори $nil : 1 \to \mathsf{List}(A)$ та $cons : A \times \mathsf{List}(A) \to \mathsf{List}(A)$ визначені як $nil = in \circ inl$ та $cons = in \circ inr$, таким чином $in = [nil, cons]$. Для кожних двох функцій $c : 1 \to C$ та $h : A \times C \to C$, катаморфізм $f = (\![c, h]\!) :$

$\mathrm{List}(A) \to C$ є унікальним розв'язком системи рівнянь:

$$\begin{cases} f \circ \mathrm{nil} = c \\ f \circ \mathrm{cons} = h \circ (\mathrm{id} \times f) \end{cases}$$

де $f = \mathrm{foldr}(c, h)$. Маючи це, ініціальна алгебра представлена функтором $\mu(1 + A \times X)$ та сумою морфізмів $[1 \to \mathrm{List}(A), A \times \mathrm{List}(A) \to \mathrm{List}(A)]$ як катаморфізму. Використовуючи це кодування, List-тип в базовій бібліотеці мови $\mathrm{O}_{\mathrm{PTS}}$ буде мати наступну форму:

$$\begin{cases} \mathrm{foldr} = (\!|[f \circ \mathrm{nil}, h]|\!), f \circ \mathrm{cons} = h \circ (\mathrm{id} \times f) \\ \mathrm{len} = (\!|[\mathrm{zero}, \lambda\ a\ n \to \mathrm{succ}\ n]|\!) \\ (++) = \lambda\ xs\ ys \to (\!|[\lambda(x) \to ys, \mathrm{cons}]|\!)(xs) \\ \mathrm{map} = \lambda\ f \to (\!|[\mathrm{nil}, \mathrm{cons} \circ (f \times \mathrm{id})]|\!) \end{cases}$$

```
data list (A: U) = cons (x: A) (cs: list A) | nil
```

$$\begin{cases} \mathrm{list} = \lambda\ \mathrm{ctor} \to \lambda\ \mathrm{cons} \to \lambda\ \mathrm{nil} \to \mathrm{ctor} \\ \mathrm{cons} = \lambda\ x\ \to \lambda\ xs \to \lambda\ \mathrm{list} \to \lambda\ \mathrm{cons} \to\ \lambda\ \mathrm{nil} \to \mathrm{cons}\ x\ (xs\ \mathrm{list}\ \mathrm{cons}\ \mathrm{nil}) \\ \mathrm{nil} = \lambda\ \mathrm{list} \to \lambda\ \mathrm{cons} \to \lambda\ \mathrm{nil} \to \mathrm{nil} \end{cases}$$

```
module list where
    map (A B: U) (f: A -> B) : list A -> list B
    length (A: U): list A -> nat
    append (A: U): list A -> list A -> list A
    foldl (A B: U) (f: B -> A -> B) (Z: B): list A -> B
    filter (A: U) (p: A -> bool) : list A -> list A
```

$$
\begin{cases}
\mathrm{len} = \mathrm{foldr}\ (\lambda\ x\ n \to \mathrm{succ}\ n)\ 0 \\
(++) = \lambda\ ys \to \mathrm{foldr}\ \mathrm{cons}\ ys \\
\mathrm{map} = \lambda\ f \to \mathrm{foldr}\ (\lambda x\ xs \to \mathrm{cons}\ (f\ x)\ xs)\ \mathrm{nil} \\
\mathrm{filter} = \lambda\ p \to \mathrm{foldr}\ (\lambda x\ xs \to \mathrm{if}\ p\ x\ \mathrm{then}\ \mathrm{cons}\ x\ xs\ \mathrm{else}\ xs)\ \mathrm{nil} \\
\mathrm{foldl} = \lambda\ f\ v\ xs = \mathrm{foldr}\ (\lambda\ xg \to\ (\lambda \to g\ (f\ a\ x)))\ \mathrm{id}\ xs\ v
\end{cases}
$$

2.7. Гомотопічна система типів HTS

```
  sys  := [ sides ]           side  := (id=0)→exp+(id=1)→exp
 form  := form\/f1+f1+f2     sides  := #empty+cos+side
  cos  := side,side+side,cos   mod  := module id where imps dec
   f1  := f1/\f2                f2  := -f2+id+0+1
  imp  := import id            brs  := #empty+cobrs
  app  := exp exp              tel  := #empty+cotel
 imps  := #list imp          cotel  := (exp:exp) tel
   id  := #list #nat           dec  := #empty+codec
   u2  := glue+unglue+Glue      u1  := fill+comp
  ids  := #list id              br  := ids→exp+ids@ids→exp
codec  := def dec
cobrs  := | br brs
  sum  := #empty+id tel+id tel|sum+id tel<ids>sys
  def  := data id tel=sum+id tel:exp=exp+id tel:exp where def
  exp  := cotel*exp+cotel→exp+exp→exp+(exp)+id
          (exp,exp)+\cotele→exp+split cobrs+exp.1+exp.2+
          ⟨ids⟩exp+exp@form+app+u2 exp exp sys+u1 exp sys
```

Тут термінали := (визначення), + (сума типів), #empty (пустий тип), #nat (тип натуральних чисел), #list (тип списків) — є частинами BNF мови. Термінали |, :, *, ⟨, ⟩, (,), =, \, /, -, →, 0, 1, @, [,], **module**, **import**, **data**, **split**, ***where***, **comp**, **fill**, **Glue**, **glue**, **unglue**, .1,

.2, а також термінал , є терміналами мови верифікатора гомотопічної системи типів. Ця мова включає в себе: індуктивні типи, вищі індуктивні типи, оператори склеювання для всесвітів та типів з відповідними елімінаторами. Усі ці концепції, та їх моделі більш формально та детально описані у наступному розділі 3.

Система не повинна бути обмежена мовами та синтаксисами, ми покажемо як приклад, підтримку гомотопічної мови з інтервалом [0,1] сумісної з cubical та з пітримкою індуктивних синтаксисів та кодувань попереднього рівня.

```
data alg
   = zero
   | one
   | max (a b: alg)
   | min (a b: alg)

data hts
   = path (t a b: lang)
   | plam (n: name) (a: alg) (b: lang)
   | papp (f: name) (a: lang) (p: alg)
   | comp_ (a b: lang)
   | fill_ (a b c: lang)
   | glue_ (a b c: lang)
   | glue_elem (a b: lang)
   | unglue_elem (a b: lang)
```

2.8. Інтерпретатор та операційна система

Мініналь мова системи O_{CPS} визначається простим синтаксичним деревом

```
data cps = var (x: nat)
         | lam (l: nat) (d: cps)
         | app (f a: cps)
```

Однак, на практиці, застосовують більш складні описи синтаксичних дерев, зокрема для лінивих обчислень, та розширення синтаксичного дерева спеціальними командами пов'язаними з середовищем виконання. Програми таких інтерпретаторів відповідно виконуються у певній пам'яті, яка використовується як контекст виконання. Кожна така програма крутиться як одиниця виконання на певному ядрі процесора. Ситема процесів, де кожен процес є CPS-програмою яку виконує інтерпретатор на певному ядрі.

Мотивація для побудови такого інтерпретатору, який повністю розміщується разом зі программою в L1 стеку (який лімітований 64КБ) базується на успіху таких віртуальних машин як LuaJIT, V8, HotSpot, а також векторних мов програмування типу K та J. Якби ми могли побудувати дійсно швидкий інтерпретатор який би виконував програми цілком в L1 кеші, байткод та стріми якого були би вирівняні по словам архітектури, а для векторних обчислень застосовувалися би AVX інструкції, які, як відомо перемагають по ціні-якості GPU обчислення. Таким чином, такий інтерпретатор міг би, навіть без спеціалізованої JIT компіляції, скласти конкуренцію сучасним промисловим інтерпретаторам, таким як Erlang, Python, K, LuaJIT.

Для дослідження цієї гіпотези мною було побудовано есперимен-тальний інтерпретатор без байт-коду, але з вирівняним по словам ар-хітерктури стріму команд, які є безпосередньою машинною презен-тацією конструкторів індуктивних типів (enum) мови Rust. Наступні результати були отримані після неотпимізованої версії інтерпретатора пнри обчисленні факторіала (5) та функції Акермана у точці (3,4).

```
Rust    0
Java    3
PyPy    8
O-CPS   291
Python  537
K       756
Erlang  10699/1806/436/9
LuaJIT  33856

akkerman_k     635 ns/iter (+/- 73)
akkerman_rust  8,968 ns/iter (+/- 322)
```

Ключовим викликом тут стали лінійні типи мови Rust, які не доз-воляють звертатися до ссилок, які вже були оброблені, а це впливає на всю архітектуру тензорного преставлення змінних в мові інтерпрета-тор CPS, яка наслідує певним чином мову K.

```
objdump ./target/release/o -d | grep mulpd
   223f1: c5 f5 59 0c d3     vmulpd (%rbx,%rdx,8),%ymm1,%ymm1
   223f6: c5 dd 59 64 d3 20  vmulpd 0x20(%rbx,%rdx,8),%ymm4,%ymm4
   22416: c5 f5 59 4c d3 40  vmulpd 0x40(%rbx,%rdx,8),%ymm1,%ymm1
   2241c: c5 dd 59 64 d3 60  vmulpd 0x60(%rbx,%rdx,8),%ymm4,%ymm4
```

```
2264d: c5 f5 59 0c d3    vmulpd (%rbx,%rdx,8),%ymm1,%ymm1
22652: c5 e5 59 5c d3 20 vmulpd 0x20(%rbx,%rdx,8),%ymm3,%ymm3
```

2.8.2. Байт-код інтерпретатора. Синтаксичне дерево, або неформалізований бай-код віртуальної машини або інтерпретатора O_{CPS} розкладається на два дерева, одне дерево для управляючих команд інтерпретатора: Defer, Continuation, Start (початок програми), Return (завершення програми).

```
data Lazy = Defer (otree: NodeId) (a: AST) (cont: Cont)
          | Continuation (otree: NodeId) (a: AST) (cont: Cont)
          | Return (a: AST)
          | Start
```

Операції віртуальної машини: умовний оператор, оператор присвоєння, лямбда функція та аплікація, є відображеннями на конструктори синтаксичного дерева.

```
data Cont = Expressions (ast: AST) (vec: Option (Iter AST)) (cont: Cont)
          | Assign (ast: AST) (cont: Cont)
          | Cond (c,d: AST) (cont: Cont)
          | Func (a,b,c: AST) (cont: Cont)
          | List (acc: Vec AST) (vec: Iter AST) (i: Nat) (cont: Cont)
          | Call (a: AST) (i: Nat) (cont: Cont)
          | Return
          | Intercore (m: Message) (cont: Cont)
          | Yield (cont: Cont)
```

2.8.3. Синтаксис. Синтаксис мови O_{CPS} підтримує тензори, та звичайне лямбда числення з значеннями у тензорах машинних типів даних: i32, i64.

```
E: V | A | C
NC: ";" = [] | ";" m:NL = m
FC: ";" = [] | ";" m:FL = m
EC: ";" = [] | ";" m:EL = m
NL: NAME | o:NAME m:NC = Cons o m
FL: E | o:E | m:FC = Cons o m
EL: E | EC  | o:E m:EC = Cons o m
C: N | c:N a:C = Call c a
N: NAME | S | HEX | L | F
L: "(" ")" = [] | "([" c:NL "]" m:FL ")" = Table c m | "(" l:EL ")" = List l
F: "{" "}" = Lambda [] [] [] | "{[" c:NL "]" m:EL "}" = Lambda [] c m
                             | "{" m:EL "}" = Lambda [] [] m
```

Після парсера, синтаксичне дерево розкладається по наступним складовим: AST для тензорів, визначення вищого рівня, Value для машинних слів, Scalar для конструкцій мови, куд входить зокрема: списки та словники, умовний оператор, присвоєння, визначення функції та її аплікація, UTF-8 літерал, та оператор передачі управління в поток планувальника який закріплений за певним ядром CPU.

```
data AST      = Atom (a: Scalar)
              | Vector (a: Vec AST)

data Value    = Nil
              | SymbolInt (a: u16)
              | SequenceInt (a: u16)
              | Number (a: i64)
              | Float (a: f64)
              | VecNumber (Vec i64)
              | VecFloat (Vec f64)

data Scalar   = Nil
              | Any
```

```
| List (a: AST)
| Dict (a: AST)
| Call (a b: AST)
| Assign (a b: AST)
| Cond (a b c: AST)
| Lambda (otree: Option NodeId) (a b: AST)
| Yield (c: Context)
| Value (v: Value)
| Name (s: String)
```

2.8.4. Операційна система. Перелічимо основні властивості операційної системи (прототип якої опублікований на Github[12]).

2.8.5. Властивості. Автобалансована низьколатентна, неблокована, без копіювання, система черг з CAS-мультикурсорами, з пріоритетами задач та масштабованими таймерами.

2.8.5.1. Асиметрична багапроцесорність. Ядро системи використовує асиметричну багапроцесорність (АП) для планування машинного часу. Так у системі для консольного вводу-виводу та вебсокет моніторингу використовується окремий ректор (закріплений за ядром процессора), аби планування не впливало на програми на інших процесорах.

Це означає статичне закріплення певного атомарного процесу обчислення за певним реактором, та навіть можливо дати гарантію, що цей процес не перерветься при наступному кванті планування ніяким іншим процесом на цьому ядрі (ситуація єдиного процесу на реактор

[12] `https://github.com/voxoz/kernel`

ядра процесору). Ядро системи постачається разом з конфігураційною мовою для закріплення задач за реакторами:

```
reactor[aux;0;mod[console;network]];
reactor[timercore;1;mod[timer]];
reactor[core1;2;mod[task]];
reactor[core2;3;mod[task]];
```

2.8.5.2. Низьколатентність. Усі реактори повинні намагатися обмежити IP-лічильник команд діапазоном розміром з L1/L2 кеш об'єм процесора, для унеможливлення колізій між ядрами на міжядерній шині можлива конфігурація, де реактори виконують код, області пам'яті якого не перетинаються, та обмежені об'ємом L1 кеш пам'яті що при наявній AVX векторизації дать змогу повністю використовувати ресурси процесору наповну.

2.8.5.3. Мультикурсори. Серцем низьколатентної системи транспорту є система наперед виділений кільцевих буферів (які називаються секторами глобального кільця). У цій системі кілець діє система курсорі для запису та читання, ці курсори можуть мати різний напрямок руху. Для забезпечення імутабельності (нерухомості даних) та відсут-

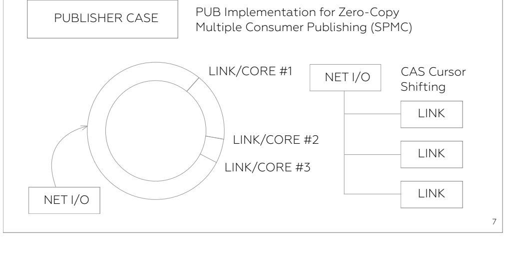

Рис. 2.3. Кільцева статична черга з CAS-курсором для публікації

ності копіювання в подальшій роботі, дані залишаються в черзі, а рухаються та передаються лише курсори на типизовані послідовності даних.

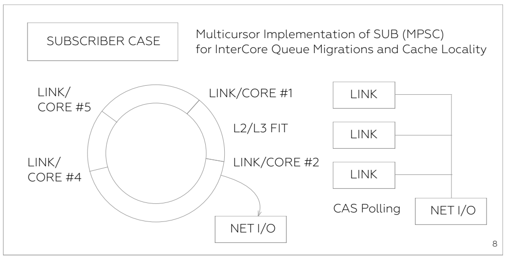

Рис. 2.4. Кільцева статична черга з CAS-курсором для згортки

2.8.5.4. Реактори. Кожен процесор має три типи реакторів які можуть бути на ньому запущені: i) Task-реактор; ii) Timer-реактор; iii) IO-цикли. Для Task-реактора існують черги пріорітетів, а для Timer-реактора — дерева інтервалів. Загальний спосіб комунікації для задач

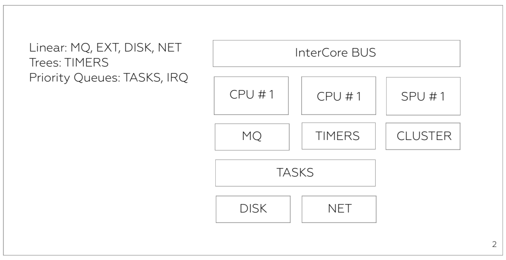

Рис. 2.5. Система процесорних ядер та реакторів

виглядає як публікація в чергу (рух курсора запису) та підписка на черги і згортання (руху курсора читання). Кожна черга має як курсори для публікації так і курсори для читання. Можливо також використання міжреакторної шини InterCore та посилання службового повідомлення по цій шині на інший реактор. Так, наприклад, працюють таймери та старти процесів, які передають сигнал в реактор для перепланування. Можна створювати нові повідомлення шини InterCore і систему фільтрів для згортання черги реактора для більш гнучкої обробки сигналів реального часу.

Task-реактор. Task-реактор або реактор задач виконує Rust задачі або програми інтерпретатора, які можуть бути двох видів: кінечні (які повертають результат виконання), або бескінечні (процеси).

Приклад бескінечної задачі — 0-процес, який запускається при старті системи. Цей процес завжди доступний по WebSocket каналу та з консолі терміналу.

IO-реактор. Мережевий сервер або IO-реактор може обслуговувати багато мережевих з’єднань та підтримує Windows, Linux, Mac смаки.

Timer-реактор. Різні типи сутностей планування (такі як Task, IO, Timer) мають різні дисципліни селекторів повідомлень для черг (послідовно, через само-балансуючі дерева, BTree дерева тощо).

2.8.5.5. Міжреакторний транспорт InterCore. Шина InterCore конструюється певним числом SPMC черг, виділених для певного ядра. Шина сама має топологію зірки між ядрами, та черга MPSC організована як функція над множиною паблішерів. Кожне ядро має рівно одного паблішера. Функція обробки шини протоколу InterCore називається poll_bus та є членом планувальника. Ви можете думати про

InterCore як телепорт між процесорами, так як pull_bus викликається після кожної операції Yield в планувальник, і, таким чином, якщо певному ядру опублікували в його чергу повідомлення, то після наступного Yield на цьому ядрі буде виконана функція обробки цього повідомлення.

`pub [ capacity ].` Створює новий CAS курсор для паблішінга, тобто для запису. Повертає глобальних машинний ідентифікатор, має єдиний параметр, розмір черги. Приклад: p: pub[16].

`sub [ publisher ].` Створює новий CAS курсор для читання певної черги, певного врайтера. Повертає глобальний машинний ідентифікатор для читання. Приклад: s: sub[p].

`spawn [ core ; program ; cursors ].` Створює нову програму задачу CPS-інтепреторатора для певного ядра. Задача може бути або програмою на мові Rust або будь якою програмою через FFI. Також при створенні задачі задається список курсорів, які ексклюзивно належатимуть до цієї задачі. Параметри функції: ядро, текст програми або назва FFI функції,спсисок курсорів. Приклад: spawn[0;”etc/proc0”;(0;1)].

`snd [ writer ; data ].` Посилає певні дані в певний курсор для запису. Повертає Nil якшо всьо ОК. Приклад: snd[p;42].

`rcv [ reader ].` Повертає прочитані дані з певного курсору. Якшо даних немає, то передає управління в планувальних за допомогою Yield. Приклад: rcv[s].

2.8.6. Структури ядра. Ядро є ситемою акторів з двома основними типами акторів: чергами, які представляють кільцеві буфери та відрізки памяті; та задачами, які резпрезентують байт-код програм та їх інтерпретацію на процесорі. Черги бувають двох видів: для публікації, які місять курсори для запису; та для читання, які містять курсори для читання. Задачі можна імплементувати як Rust програми, або як O_{CPS} програми.

```
pub struct Publisher<T> {
    ring: Arc<RingBuffer<T>>,
    next: Cell<Sequence>,
    cursors: UncheckedUnsafeArc<Vec<Cursor>>,
}
```

```
pub struct Subscriber<T> {
    ring: Arc<RingBuffer<T>>,
    token: usize,
    next: Cell<Sequence>,
    cursors: UncheckedUnsafeArc<Vec<Cursor>>,
}
```

Існує дві спеціільні задачі: InterCore задача, написана на Rust, та запускається на всіх ядрах при запуску системи, а також CPS-інтерпретор головного термінала системи, цей інтерпретор запускається на BSP ядрі, поближче до Console та WebSocket IO селекторів. В процесі життя різні CPS та Rust задачі можуть бути запущені в такій системі, поєднуючи гнучкість програм інтерпретатора, та низькорівненивих програм написаних на мові Rust.

Окрім черг та задач, в системі присутні також таймери та інші IO задачі, такі як сервери мережі або сервери доступу до файлів. Також існують структури які репрезентують ядра та містять палнувальники. Уся віртуальна машина є сукупністю таких структур-ядер.

2.8.6.3. Канал. Канал складається з одного курсору для запису та багатьох курсорів для читання. Канал предствляє собою компонент зірки шини InterCore.

```
pub struct Channel {
    publisher: Publisher<Message>,
    subscribers: Vec<Subscriber<Message>>,
}
```

2.8.6.4. Черги ядра. Память репрезентує усі наявні черги для публікації та читання на ядрі. Ця інформація передається клонованою кожній задачі планувальника на цьому ядрі.

```
pub struct Memory<'a> {
    publishers: Vec<Publisher<Value<'a>>,
    subscribers: Vec<Subscriber<Value<'a>>>,
}
```

2.8.6.5. Планувальник. Планувальник репрезентує ядро процесора, які розрізняються як BSP-ядра (або 0-ядра, bootstrap) та AP ядра (інші ядра > 0, application). BSP ядро тримає на собі Console та WebSocket IO селектори. Це означає, що BSP ядро дає свій час на обробку зовнішньої інформації, у той час як AP процесори не обтяжені таким навантаженням (io черга в таких планувальниках пуста). Існує

InterCore повідомлення яке додає або видаляє довільні IO селектори в планувальних для довільних конфігурацій.

```
pub struct Scheduler<'a> {
    pub tasks: Vec<T3<Job<'a>>>,
    pub bus: Channel,
    pub queues: Memory<'a>,
    pub io: IO,
}
```

2.8.7. Протокол InterCore. Протокол шини InterCore.

```
pub enum Message {
    Pub(Pub),
    Sub(Sub),
    Print(String),
    Spawn(Spawn),
    AckSub(AckSub),
    AckPub(AckPub),
    AckSpawn(AckSpawn),
    Exec(usize, String),
    Select(String, u16),
    QoS(u8, u8, u8),
    Halt,
    Nop,
}
```

2.9. Висновки

Як апогей, система HTS є фінальною категорією, куди сходяться всі стрілки категорії мов. Кожна мова та її категорія мають певний набір стрілок ендоморфізмів, які обчислюють, верифікують, нормалізують, оптимізують програми своїх мов. Стрілки виду $e_i : O_{n+1} \to O_n$ є екстракторами, які понижають систему типів, при чому $O_{CPS} = O_0$.

Базова бібліотека мови Ерланг в яку проводиться основний естракт йде з дистрибутивом Erlang/OTP. Базова бібліотека O_{PTS} наведена в репозиторії Github[13]. Гомотопічна базова бібліотека відповідає термінальній мові O_{CCHM}, та теж відкрита на Github[14]. Останні два розділи присвячені математичному моделюванню математики на цій мові.

2.9.1. Базова бібліотека. Перша частина гомотопічної базової бібліотеки — це основи гомотопічної теорії типів, з основними визначеннями та теоремами.

2.9.2. Математичні компоненти. Друга частина гомотопічної базової бібліотеки — це формалізація математики, як приклад використання розробленої концепцтуальної моделі системи доведення теорем.

[13] `https://github.com/groupoid/om`

[14] `https://github.com/groupoid/infinity`

РОЗДІЛ 3
БАЗОВА БІБЛІОТЕКА

В третьому розділі дається опис гомотопічної мови програмування, реалізація якої вперше була представлена CCHM в 2017 році, та для якої написана гомотопічна базова бібліотека представлена у цьому та наступному розділах.

Опис гомотопічної мови дається у розрізі індуктивних типів, та показується, що вищі індуктивні типи з вищою рівністю є аналогом CW-комплексів, та розширюють гомотопічну теорію типів (без UIP) так само, як індуктивні типи є аналогом поліноміальних функторів та розширюю теорію типів Мартіна-Льофа. Усі ці чотири частини, а також секція Модальності складають основу гомотопічної мови програмування, та не містить складної математики (окрім опису інтерпретацій, які є матаметичними ізморізмами).

3.1. Інтерналізація теорії типів

Each language implementation needs to be checked. The one of possible test cases for type checkers is the direct embedding of type theory model into the language of type checker. As types in Martin-Löf Type Theory (MLTT) are formulated using 5 types of rules (formation, introduction, elimination, computation, uniqueness), we construct aliases for host language primitives and use type checker to prove that it is MLTT. This could be seen as ultimate test sample for type checker as intro-elimination fusion resides in beta-eta rules, so by proving them we prove properties of the host type checker.

Also this issue opens a series of articles dedicated to formalization in cubical type theory the foundations of mathematics. This issue is dedicated to MLTT modeling and its verification. Also as many may not be familiar with Π and Σ types, this issue presents different interpretation of MLTT types.

3.1.0.1. Теорія типів. MLTT could be reduced to Π, Σ, Path types, as W-types could be modeled through Σ and Fin/Nat/List/Maybe types could be modeled on W. In this issue Π, Σ, Path are given as a core MLTT and W-types are given as exercise. List, Nat, Fin types are defined in next section.

Any new type in MLTT presented with set of 5 rules: i) formation rules, the signature of type; ii) the set of constructors which produce the elements of formation rule signature; iii) the dependent eliminator or induction principle for this type; iv) the beta-equality or computational rule; v) the eta-equality or uniqueness principle. Π, Σ, and Path types will be given shortly. This interpretation or rather way of modeling is MLTT specific.

The most interesting are Id types. Id types were added in [1]1984 while original MLTT was introduced in [2]1972. Predicative Universe Hierarchy was added in [3]1975. While original MLTT contains Id types that preserve

[1]P. Martin-Löf, G. Sambin. Intuitionistic type theory. 1984.

[2]P. Martin-Löf, G. Sambin. The Theory of Types. 1972.

[3]P. Martin-Löf. An intuitionistic theory of types: predicative part. 1975.

Таблиця 3.1

Interpretations correspond to mathematical theories

Type Theory	Logic	Category Theory	Homotopy Theory
A type	class	object	space
isProp A	proposition	(-1)-truncated object	space
a:A program	proof	generalized element	point
$B(x)$	predicate	indexed object	fibration
$b(x) : B(x)$	conditional proof	indexed elements	section
$\varnothing$	$\perp$ false	terminal object	empty space
$\mathbf{1}$	$\top$ true	initial object	singleton
$A + B$	$A \vee B$ disjunction	coproduct	coproduct space
$A \times B$	$A \wedge B$ conjunction	product	product space
$A \to B$	$A \Rightarrow B$	internal hom	function space
$\sum x : A, B(x)$	$\exists_{x:A} B(x)$	dependent sum	total space
$\prod x : A, B(x)$	$\forall_{x:A} B(x)$	dependent product	space of sections
$\mathbf{Path}_A$	equivalence $=_A$	path space object	path space A^I
quotient	equivalence class	quotient	quotient
W-type	induction	colimit	complex
type of types	universe	object classifier	universe
quantum circuit	proof net	string diagram	

uniquness of identity proofs (UIP) or eta-rule of Id type, HoTT refutes UIP (eta rule desn't hold) and introduces univalent heterogeneous Path equality ([4]∞-Groupoid interpretation). Path types are essential to prove computation and uniquness rules for all types (needed for building signature and terms), so we will be able to prove all the MLTT rules as a whole.

3.1.0.2. Інтерпретації. In contexts you can bind to variables (through de Brujin indexes or string names): i) indexed universes; ii) built-in types; iii) user constructed types, and ask questions about type derivability, type checking and code extraction. This system defines the core type checker within its language.

By using this languages it is possible to encode different interpretations of type theory itself and its syntax by construction. Usually the issues will refer to following interpretations: i) type-theoretical; ii) categorical; iii) set-theoretical; iv) homotopical; v) fibrational or geometrical.

Логічна або теоретико-типова інтерпретація. According to type theoretical interpretation for any type should be provided 5 formal inference rules: i) formation; ii) introduction; iii) dependent elimination principle; iv) beta rule or computational rule; v) eta rule or uniqueness rule. The last one could be exceptional for Path types. The formal representation of all rules of MLTT are given according to type-theoretical interpretation as a final result in this Issue I. It was proven that classical Logic could be embedded into intuitionistic propositional logic (IPL) which is directly embedded into

[4]M. Hofmann, T. Streicher. The groupoid interpretation of type theory. 1996.

MLTT.

Logical and type-theoretical interpretations could be distincted. Also set-theoretical interpretation is not presented in Table 1.

Категоріальна або топосо-теоритична інтерпретація. Categorical interpretation is a modeling through categories and functors. First category is defined as objects, morphisms and their properties, then we define functors, etc. In particular, as an example, according to categorical interpretation Π and Σ types of MLTT are presented as adjoint functors, and forms itself a locally closed cartesian category, which will be given a intermediate result in **Issue VII: Topos Theory**. In some sense we include here topos-theoretical interpretations, with presheaf model of type theory as example (in this case fibrations are constructes as functors, categorically).

Теоретико-типова інтерпертація. Set-theoretical interpretations could replace first-order logic, but could not allow higher equalities, as long as inductive types to be embedded directly. Set is modelled in type theory according to homotopical interpretation as n-type.

Гомотопічна інтерпретація. In classical MLTT uniquness rule of Id type do holds strictly. In Homotopical interpretation of MLTT we need to allow a path space as Path type where uniqueness rule doesn’t hold. Groupoid interpretation of Path equality that doesn’t hold UIP generally was given in 1996 by Martin Hofmann and Thomas Streicher.

When objects are defined as fibrations, or dependent products, or indexed-objects this leds to fibrational semantics and geometric sheaf interpretation. Several definition of fiber bundles and trivial fiber bindle as direct isomorphisms of Π types is given here as theorem. As fibrations study in homotopical interpretation, geometric interpretation could be treated as homotopical.

3.1.1. Типи Π, Σ, Path.

3.1.1.1. Π-тип. Π is a dependent product type, the generalization of functions. As a function it can serve the wide range of mathematical constructions as its domain and codomain, which are in general: objects, types, or spaces; and could have as its instance: sets, functions, polynomial functors, infinitesimals, ∞-groupoids, topological ∞-groupoid, CW-complexes, categories, languages, etc.

At this light there could be many interpretation of Π types from different areas of mathematics. We give here three: i) logical interpretation of Π as $\forall$ quantifier from higher order logic that forms a ground of type theory; ii) geomeric intepretation of Π as fiber bundle; iii) categorical interpretation of functions as functors.

Теоретико-типова інтерпретація. As a logical system dependent type theory could correspond to higher order logic. However here only type-theoretical model is given completely.

Визначення 30. (П-Formation).

$$(x : A) \to B(x) =_{def} \prod_{x:A} B(x) : U.$$

```
Pi (A: U) (B: A -> U): U = (x: A) -> B x
```

Визначення 31. (П-Introduction).

$$\backslash(x : A) \to b(x) =_{def} \prod_{A:U} \prod_{B:A\to U} \prod_{b:\prod_{a:A} B(a)} \lambda x.b(x) : \prod_{y:A} B(y).$$

```
lambda (A B: U) (b: B): A -> B = \(x: A) -> b
lam (A: U) (B: A -> U) (b: (a: A) -> B a): Pi A B = \(x: A) -> b x
```

Визначення 32. (П-Elimination).

$$f\ a =_{def} \prod_{A:U} \prod_{B:A\to U} \prod_{a:A} \prod_{f:\prod_{x:A} B(a)} f(a) : B(a).$$

```
apply (A B: U) (f: A -> B) (a: A) : B = f a
app (A: U) (B: A -> U) (a: A) (f: Pi A B): B a = f a
```

Теорема 1. (П-Computation).

$$f(a) =_{B(a)} (\lambda(x : A) \to f(a))(a).$$

```
Beta (A:U) (B:A->U) (a: A) (f: Pi A B)
   : Path (B a) (app A B a (lam A B f)) (f a)
```

Теорема 2. (Π-Uniqueness).

$$f =_{(x:A)\to B(a)} (\lambda(y : A) \to f(y)).$$

```
Eta (A:U) (B:A->U) (a:A) (f: Pi A B)
   : Path (Pi A B) f (\(x:A) -> f x)
```

Категоріальна інтерпретація. The adjoints Π and Σ is not the only adjoints could be presented in type system. Axiomatic cohesions could contain a set of adjoint pairs as a core type checker operations.

Визначення 33. (Dependent Product). The dependent product along morphism $g : B \to A$ in category C is the right adjoint $\Pi_g : C_{/B} \to C_{/A}$ of the base change functor.

Визначення 34. (Space of Sections). Let $\mathbf{H}$ be a $(\infty, 1)$-topos, and let $E \to B : \mathbf{H}_{/B}$ a bundle in $\mathbf{H}$, object in the slice topos. Then the space of sections $\Gamma_\Sigma(E)$ of this bundle is the Dependent Product:

$$\Gamma_\Sigma(E) = \Pi_\Sigma(E) \in \mathbf{H}.$$

Теорема 3. (HomSet). If codomain is set then space of sections is a set.

```
setFun (A B : U) (_: isSet B) : isSet (A -> B)
```

Теорема 4. (Contractability). If domain and codomain is contractible then the space of sections is contractible.

```
piIsContr (A: U) (B: A -> U) (u: isContr A)
          (q: (x: A) -> isContr (B x)) : isContr (Pi A B)
```

Визначення 35. (Section). A section of morphism $f : A \to B$ in some category is the morphism $g : B \to A$ such that $f \circ g : B \xrightarrow{g} A \xrightarrow{f} B$ equals the identity morphism on B.

Гомотопічна інтерпретація. Geometrically, Π type is a space of sections, while the dependent codomain is a space of fibrations. Lambda functions are sections or points in these spaces, while the function result is a fibration. Π type also represents the cartesian family of sets, generalizing the cartesian product of sets.

Визначення 36. (Fiber). The fiber of the map $p : E \to B$ in a point $y : B$ is all points $x : E$ such that $p(x) = y$.

Визначення 37. (Fiber Bundle). The fiber bundle $F \to E \xrightarrow{p} B$ on a total space E with fiber layer F and base B is a structure (F, E, p, B) where $p : E \to B$ is a surjective map with following property: for any point $y : B$ exists a neighborhood U_b for which a homeomorphism $f : p^{-1}(U_b) \to U_b \times F$ making the following diagram commute.

$$\begin{tikzcd} p^{-1}(U_b) \arrow[r, "f"] \arrow[d, "p"'] & U_b \times F \arrow[dl, "pr_1"] \\ U_b & \end{tikzcd}$$

Визначення 38. (Cartesian Product of Family over B). Is a set F of sections of the bundle with elimination map $app : F \times B \to E$ such that

$$F \times B \xrightarrow{app} E \xrightarrow{pr_1} B \qquad (3.1)$$

pr_1 is a product projection, so pr_1, app are morphisms of slice category $Set_{/B}$. The universal mapping property of F: for all A and morphism $A \times B \to E$ in $Set_{/B}$ exists unique map $A \to F$ such that everything commute. So a category with all dependent products is necessarily a category with all pullbacks.

Визначення 39. (Trivial Fiber Bundle). When total space E is cartesian product $\Sigma(B, F)$ and $p = pr_1$ then such bundle is called trivial $(F, \Sigma(B, F), pr_1, B)$.

Теорема 5. (Functions Preserve Paths). For a function $f : (x : A) \to B(x)$ there is an $ap_f : x =_A y \to f(x) =_{B(x)} f(y)$. This is called application of f to path or congruence property (for non-dependent case — $cong$ function). This property behaves functoriality as if paths are groupoid morphisms and types are objects.

Теорема 6. (Trivial Fiber equals Family of Sets). Inverse image (fiber) of fiber bundle $(F, B * F, pr_1, B)$ in point $y : B$ equals $F(y)$.

```
FiberPi (B: U) (F: B -> U) (y: B)
      : Path U (fiber (Sigma B F) B (pi1 B F) y) (F y)
```

Теорема 7. (Homotopy Equivalence). If fiber space is set for all base, and there are two functions $f, g : (x : A) \to B(x)$ and two homotopies between them, then these homotopies are equal.

```
setPi (A: U) (B: A -> U) (h: (x: A) -> isSet (B x)) (f g: Pi A B)
      (p q: Path (Pi A B) f g) : Path (Path (Pi A B) f g) p q
```

Note that we will not be able to prove this theorem until **Issue III: Homotopy Type Theory** because bi-invertible iso type will be announced there.

3.1.1.2. Σ-тип. Σ is a dependent sum type, the generalization of products. Σ type is a total space of fibration. Element of total space is formed as a pair of basepoint and fibration.

3.1.1.3. Теоретико-типов інтерпретація.

Визначення 40. (Σ-Formation).

```
Sigma (A : U) (B : A -> U) : U = (x : A) * B x
```

Визначення 41. (Σ-Introduction).

```
dpair (A: U) (B: A -> U) (a: A) (b: B a) : Sigma A B = (a,b)
```

Визначення 42. (Σ-Elimination).

```
pr1 (A: U) (B: A -> U)
    (x: Sigma A B): A = x.1

pr2 (A: U) (B: A -> U)
    (x: Sigma A B): B (pr1 A B x) = x.2

sigInd (A: U) (B: A -> U) (C: Sigma A B -> U)
       (g: (a: A) (b: B a) -> C (a, b))
       (p: Sigma A B) : C p = g p.1 p.2
```

Теорема 8. (Σ-Computation).

```
Beta1 (A: U) (B: A -> U)
      (a:A) (b: B a)
    : Equ A a (pr1 A B (a,b))

Beta2 (A: U) (B: A -> U)
      (a: A) (b: B a)
    : Equ (B a) b (pr2 A B (a,b))
```

Теорема 9. (Σ-Uniqueness).

```
Eta2 (A: U) (B: A -> U) (p: Sigma A B)
   : Equ (Sigma A B) p (pr1 A B p,pr2 A B p)
```

3.1.1.4. Категоріальна інтерпретація.

Визначення 43. (Dependent Sum). The dependent sum along the morphism $f : A \to B$ in category C is the left adjoint $\Sigma_f : C_{/A} \to C_{/B}$ of the base change functor.

3.1.1.5. Теоретико-множинна інтерпретація.

Теорема 10. (Axiom of Choice). If for all $x : A$ there is $y : B$ such that $R(x, y)$, then there is a function $f : A \to B$ such that for all $x : A$ there is a witness of $R(x, f(x))$.

```
ac (A B: U) (R: A -> B -> U)
 : (p: (x:A) -> (y:B)*(R x y)) -> (f:A->B) * ((x:A)->R(x)(f x))
```

Теорема 11. (Total). If fiber over base implies another fiber over the same base then we can construct total space of section over that base with another fiber.

```
total (A:U) (B C: A -> U)
      (f: (x:A) -> B x -> C x) (w: Sigma A B)
    : Sigma A C = (w.1,f (w.1) (w.2))
```

Теорема 12. (Σ-Contractability). If the fiber is set then the Σ is set.

```
setSig (A:U) (B: A -> U) (sA: isSet A)
       (sB : (x:A) -> isSet (B x)) : isSet (Sigma A B)
```

Теорема 13. (Path Between Sigmas). Path between two sigmas $t, u : \Sigma(A, B)$ could be decomposed to sigma of two paths $p : t_1 =_A u_1)$ and $(t_2 =_{B(p@i)} u_2)$.

```
pathSig (A:U) (B : A -> U) (t u : Sigma A B)
      : Path U (Path (Sigma A B) t u)
               ((p: Path A t.1 u.1) * PathP (<i>B(p@i)) t.2 u.2)
```

3.1.1.6. Path-тип. The Path identity type defines a Path space with elements and values. Elements of that space are functions from interval $[0, 1]$ to a values of that path space. This ctt file reflects [5]CCHM cubicaltt model with connections. For [6]ABCFHL yacctt model with variables please refer to ytt file. You may also want to read [7]BCH, [8]AFH. There is a [9]PO paper about CCHM axiomatic in a topos.

3.1.1.7. Кубічна інтерпретація.

Визначення 44. (Path Formation).

```
Hetero (A B: U) (a: A) (b: B) (P: Path U A B) : U = PathP P a b
Path (A: U) (a b: A) : U = PathP (<i> A) a b
```

Визначення 45. (Path Reflexivity). Returns an element of reflexivity path space for a given value of the type. The inhabitant of that path space is the lambda on the homotopy interval $[0, 1]$ that returns a constant value a. Written in syntax as `<i>a` which equals to $\lambda\ (i : I) \to a$.

```
refl (A: U) (a: A) : Path A a a
```

Визначення 46. (Path Application). You can apply face to path.

[5]Cyril Cohen, Thierry Coquand, Simon Huber, Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. 2015. https://5ht.co/cubicaltt.pdf

[6]Carlo Angiuli, Brunerie, Coquand, Kuen-Bang Hou (Favonia), Robert Harper, Dan Licata. Cartesian Cubical Type Theory. 2017. https://5ht.co/cctt.pdf

[7]Marc Bezem, Thierry Coquand, Simon Huber. A model of type theory in cubical sets. 2014. http://www.cse.chalmers.se/~coquand/mod1.pdf

[8]Carlo Angiuli, Kuen-Bang Hou (Favonia), Robert Harper. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. 2018. https://www.cs.cmu.edu/~cangiuli/papers/ccctt.pdf

[9]Andrew Pitts, Ian Orton. Axioms for Modelling Cubical Type Theory in a Topos. 2016. https://arxiv.org/pdf/1712.04864.pdf

```
app1 (A: U) (a b: A) (p: Path A a b): A = p @ 0
app2 (A: U) (a b: A) (p: Path A a b): A = p @ 1
```

Визначення 47. (Path Composition). Composition operation allows to build a new path by given to paths in a connected point.

$$\begin{tikzcd}
a \arrow[r, "comp"] & c \\
a \arrow[u, "\lambda(i:I) \to a"] \arrow[r, "p@i"] & b \arrow[u, "q"']
\end{tikzcd}$$

```
composition (A: U) (a b c: A) (p: Path A a b) (q: Path A b c)
          : Path A a c = comp (<i>Path A a (q@i)) p []
```

Теорема 14. (Path Inversion).

```
inv (A: U) (a b: A) (p: Path A a b): Path A b a = <i> p @ -i
```

Визначення 48. (Connections). Connections allows you to build square with given only one element of path: i) $\lambda\ (i,j:I) \to p\ @\ min(i,j)$; ii) $\lambda\ (i,j:I) \to p\ @\ max(i,j)$.

$$\begin{tikzcd}
a \arrow[r, "p"] & b \\
a \arrow[u, "\lambda\ (i:I) \to a"] \arrow[r, "\lambda\ (i:I) \to a"] & a \arrow[u, "p"']
\end{tikzcd}
\qquad
\begin{tikzcd}
b \arrow[r, "\lambda\ (i:I) \to b"] & b \\
a \arrow[u, "p"] \arrow[r, "p"] & b \arrow[u, "\lambda\ (i:I) \to b"']
\end{tikzcd}$$

```
connection1 (A: U) (a b: A) (p: Path A a b)
          : PathP (<x> Path A (p@x) b) p (<i>b)
          = <y x> p @ (x \/ y)

connection2 (A: U) (a b: A) (p: Path A a b)
          : PathP (<x> Path A a (p@x)) (<i>a) p
```

```
        = <x y> p @ (x /\ y)
```

Теорема 15. (Congruence). Is a map between values of one type to path space of another type by an encode function between types. Implemented as lambda defined on $[0, 1]$ that returns application of encode function to path application of the given path to lamda argument |λ (i:I) → f (p @ i)| for both cases.

```
ap   (A B: U) (f: A -> B)
     (a b: A) (p: Path A a b)
   : Path B (f a) (f b)

apd (A: U) (a x:A) (B: A -> U) (f: A -> B a)
     (b: B a) (p: Path A a x)
   : Path (B a) (f a) (f x)
```

Теорема 16. (Transport). Transports a value of the domain type to the value of the codomain type by a given path element of the path space between domain and codomain types. Defined as path composition with |[]| of a over a path p — |comp p a []|.

```
trans (A B: U) (p: Path U A B) (a: A) : B
```

3.1.1.8. Теоретико-типова інтерпретація.

Визначення 49. (Singleton).

```
singl (A: U) (a: A): U = (x: A) * Path A a x
```

Теорема 17. (Singleton Instance).

```
eta (A: U) (a: A): singl A a = (a,refl A a)
```

Теорема 18. (Singleton Contractability).

```
contr (A: U) (a b: A) (p: Path A a b)
  : Path (singl A a) (eta A a) (b,p)
  = <i> (p @ i,<j> p @ i/\j)
```

Теорема 19. (Path Elimination, Diagonal).

```
D (A: U) : U = (x y: A) -> Path A x y -> U
J (A: U) (x y: A) (C: D A)
  (d: C x x (refl A x))
  (p: Path A x y) : C x y p
= subst (singl A x) T (eta A x) (y, p) (contr A x y p) d where
  T (z: singl A x) : U = C x (z.1) (z.2)
```

Теорема 20. (Path Elimination, Paulin-Mohring). J is formulated in a form of Paulin-Mohring and implemented using two facts that singleton are contractible and dependent function transport.

```
J (A: U) (a b: A)
  (P: singl A a -> U)
  (u: P (a,refl A a))
  (p: Path A a b) : P (b,p)
```

Теорема 21. (Path Elimination, HoTT). J from HoTT book.

```
J (A: U) (a b: A)
  (C: (x: A) -> Path A a x -> U)
  (d: C a (refl A a))
  (p: Path A a b) : C b p
```

Теорема 22. (Path Computation).

```
trans_comp (A: U) (a: A)
  : Path A a (trans A A (<_> A) a)
  = fill (<i> A) a []
subst_comp (A: U) (P: A -> U) (a: A) (e: P a)
  : Path (P a) e (subst A P a a (refl A a) e)
  = trans_comp (P a) e
```

```
J_comp (A: U) (a: A) (C: (x: A) -> Path A a x -> U) (d: C a (refl A a))
  : Path (C a (refl A a)) d (J A a C d a (refl A a))
  = subst_comp (singl A a) T (eta A a) d where T (z: singl A a)
  : U = C a (z.1) (z.2)
```

Note that Path type has no Eta rule due to groupoid interpretation.

3.1.1.9. Групоїдна інтерпретація. The groupoid interpretation of type theory is well known article by Martin Hoffman and Thomas Streicher, more specific interpretation of identity type as infinity groupoid. The groupoid interpretation of Path equality will be given along with category theory library in **Issue VII: Category Theory**.

3.1.2. Всесвіти. This introduction is a bit wild strives to be simple yet precise. As we defined a language BNF we could define a language AST by using inductive types which is yet to be defined in **Issue II: Inductive Types and Models**. This SAR notation is due Barendregt.

Визначення 50. (Terms). Point in initial object of language AST inductive definition is called a term. If type theory or language is defined as an inductive type (AST) then the term is defined as its instance.

Визначення 51. (Sorts). N-indexed set of universes $U_{n \in N}$. Could have any number of elements which defines different type systems. All built-in types as long as user defined types are landed usually by default in U_0 universe. Sorts represented in type checker as a separate constructor.

Визначення 52. (Axioms). The inclusion rules $U_i : U_j, i, j \in N$, that define which universe is element of another given universe. You may attach any rules that joins i, j in some way. Axioms with sorts define universe hierarchy.

Визначення 53. (Rules). The set of landings $U_i \to U_j : U_{\smile(i,j),i,j \in N}$, where $\smile : N \times N \to N$. These rules define term dependence or how we land (in which universe) formation rules in definitions.

Визначення 54. (Predicative hierarchy). If $\smile$ in Rules is an uncurried function $\max : N \times N \to N$ then such universe hierarchy is called predicative.

Визначення 55. (Impredicative hierarchy). If λ in Rules is a second projection of a tuple $\mathrm{snd} : N \times N \to N$ then such universe hierarchy is called impredicative.

Визначення 56. (Definitional Equality). For any $U_i, i \in N$ there is defined an equality between its members and between its instances. For all x,y $\in$ A, there is defined a x=y. Definitional equality compares normalized term instances.

Визначення 57. (SAR). The universum space is configured with a triple of: i) sorts, a set of universes $U_{n \in N}$ indexed over set N; ii) axioms, a set of inclusions $U_i : U_j, i, j \in N$; iii) rules of term dependence universe landing, a set of landings $U_i \to U_j : U_{\smile(i,j),i,j \in N}$, where λ could be function max (predicative) or snd (impredicative).

Приклад 1. (CoC). SAR = $\{\{\star, \Box\}, \{\star : \Box\}, \{i \to j : j; i, j \in \{\star, \Box\}\}$. Terms live in universe $\star$, and types live in universe $\Box$. In CoC $\smile = \mathrm{snd}$.

Приклад 2. (PTS^{∞}). SAR $= \{U_{i \in N}, U_i : U_{j;i<j;i,j \in N}, U_i \to U_j : U_{\smile(i,j);i,j \in N}\}$. Where U_i is a universe of i-level or i-category in categorical interpretation. The working prototype of PTS^{∞} is given in **Addendum I:**

Pure Type System for Erlang[10].

3.1.3. Контексти.

Speaking of type checker execution, we introduce context or dictionary with types and terms, from which we can derive typed variables. This chain could be implemented as nested sigma types (due to R.A.G.Seely) or list types (due to Voevodsky). Categorically dependent type theory is built upon categories of contexts.

Визначення 58. (Empty Context).

$$\gamma_0 : \Gamma =_{\mathrm{def}} \star.$$

Визначення 59. (Context Comprehension).

$$\Gamma \,; A =_{\mathrm{def}} \sum_{\gamma:\Gamma} A(\gamma).$$

Визначення 60. (Context Derivability).

$$\Gamma \vdash A =_{\mathrm{def}} \prod_{\gamma:\Gamma} A(\gamma).$$

3.1.4. Інтерналізація.

Here is given formal model of type-theoretical interpretation of Martin-Löf Type Theory. It combines 4 Path rules (no eta), 5 Π rules, and 6 Σ rules (two elims). The proof is provided by direct embedding (internalizing) the model intro the model of type checker which is even more powerful.

---

[10] M.Sokhatsky,P.Maslianko. The Systems Engineering of Consistent Pure Language with Effect Type System for Certified Applications and Higher Languages. AIP Conference Proceedings. 2018. doi:10.1063/1.5045439

Визначення 61. (MLTT). The MLTT as a Type is defined by taking all rules for Π, Σ and Path types into one Σ telescope or context.

```
MLTT (A: U): U
  = (Pi_Former: (A -> U) -> U)
  * (Pi_Intro: (B: A -> U) -> ((a: A) -> B a) -> (Pi A B))
  * (Pi_Elim: (B: A -> U) (a: A) -> (Pi A B) -> B a)
  * (Pi_Comp1: (B: A -> U) (a : A) (f: Pi A B) ->
    Equ (B a) (Pi_Elim B a (Pi_Intro B f)) (f a))
  * (Pi_Comp2: (B: A -> U) (a: A) (f: Pi A B) ->
    Equ (Pi A B) f (\(x:A) -> f x))
  * (Sigma_Former: (A -> U) -> U)
  * (Sigma_Intro: (B: A -> U) (a: A) -> (b: B a) -> Sigma A B)
  * (Sigma_Elim1: (B: A -> U) (_: Sigma A B) -> A)
  * (Sigma_Elim2: (B: A -> U) (x: Sigma A B) -> B (pr1 A B x))
  * (Sigma_Comp1: (B: A -> U) (a: A) (b: B a) ->
    Path A a (Sigma_Elim1 B (Sigma_Intro B a b)))
  * (Sigma_Comp2:  (B: A -> U) (a: A) (b: B a) ->
    Path (B a) b (Sigma_Elim2 B (a,b)))
  * (Sigma_Comp3: (B: A -> U) (p: Sigma A B)  ->
    Path (Sigma A B) p (pr1 A B p,pr2 A B p))
  * (Id_Former: A -> A -> U)
  * (Id_Intro: (a: A) -> Path A a a)
  * (Id_Elim: (x: A) (C: D A) (d: C x x (Id_Intro x))
    (y: A) (p: Path A x y) -> C x y p)
  * (Id_Comp: (a:A)(C: D A) (d: C a a (Id_Intro a)) ->
    Path (C a a (Id_Intro a)) d (Id_Elim a C d a (Id_Intro a))) * U
```

Теорема 23. (Model Check). There is an instance of MLTT.

```
instance (A: U): MLTT A
    = (Pi A, lam A, app A, Beta A, Eta A,
       Sigma A, dpair A, pr1 A, pr2 A, Beta1 A, Beta2 A, Eta2 A,
       Path A, refl A, J A, J_comp A, A)
```

Перевірка в кубічній теорії. The result of the work is a `mltt.ctt` file which can be runned using `cubicaltt`. Note that computation rules take a seconds to type check.

```
\$ time cubical -b mltt.ctt
Checking: MLTT
Checking: instance
File loaded.

real    0m6.308s
user    0m6.278s
sys     0m0.014s
```

3.2. Індуктивні типи

3.2.1. Empty.

empty type lacks both introduction rules and eliminators. However, it has recursor and induction.

```
data empty =
emptyRec (C: U): empty -> C = split {}
emptyInd (C: empty -> U): (z: empty) -> C z = split {}

data unit = star
unitRec (C: U) (x: C): unit -> C = split tt -> x
unitInd (C: unit -> U) (x: C tt): (z: unit) -> C z = split tt -> x
```

3.2.3. Bool.

Визначення 62. (Bool). bool is a run-time version of the boolean logic you may use in your general purpose applications. bool is isomorphic to 1+1: either unit unit.

```
data bool = false | true
b1: U = bool -> bool
b2: U = bool -> bool -> bool
negation: b1 = split { false -> true; true -> false }
or: b2 = split { false -> idfun bool; true -> lambda bool bool true }
and: b2 = split { false -> lambda bool bool false; true -> idfun boo }
boolEq: b2 = lamb bool (bool -> bool) negation
boolRec (C: U) (f t: C): bool -> C = split { false -> f ; true -> t }
boolInd (C: bool -> U) (f: A false) (t: A true): (n:bool) -> A n
  = split { false -> f ; true -> t }
```

3.2.4. Maybe.

Визначення 63. (Maybe). Maybe has representing functor $M_A(X) = 1 + A$. It is used for wrapping values with optional nothing constructor. In ML-family languages this type is called Option (Miranda, ML). There is an isomorphims between (fix maybe) and nat.

```
data maybe (A: U) = nothing | just (x: A)
maybeRec (A P: U) (n: P) (j: A -> P): maybe A -> P
       = split { nothing -> n; just a -> j a }

maybeInd (A: U) (P: maybe A -> U) (n: P nothing)
         (j: (a: A) -> P (just a)): (a: maybe A) -> P a
       = split { nothing -> n ; just x -> j x }
```

3.2.5. Either.

either is a representation for sum types or disjunction.

```
data either (A B: U) = left (x: A) | right (y: B)
eitherRec (A B C: U) (b: A -> C) (c: B -> C): either A B -> C
        = split { inl x -> b(x) ; inr y -> c(y) }

eitherInd (A B: U) (C: either A B -> U)
          (x: (a: A) -> C (inl a))
```

```
            (y: (b: B) -> C (inr b))
          : (x: either A B) -> C x
          = split { inl i -> x i ; inr j -> y j }
```

3.2.6. Tuple. tuple is a representation for non-dependent product types or conjunction.

```
data tuple (A B: U) = pair (x: A) (y: B)
prod (A B: U) (x: A) (y: B): (_: A) * B = (x,y)
tupleRec  (A B C: U) (c: (x:A) (y:B) -> C): (x: tuple A B) -> C
        = split pair a b -> c a b
tupleInd  (A B: U) (C: tuple A B -> U)
          (c: (x:A)(y:B) -> C (pair x y))
        : (x: tuple A B) -> C x
        = split pair a b -> c a b
```

3.2.7. Nat. Pointed Unary System is a category nat with the terminal object and a carrier nat having morphism [zero: 1nat → nat, succ: nat → nat]. The initial object of nat is called Natural Number Object and models Peano axiom set.

```
data nat = zero | succ (n: nat)
natEq: nat -> nat -> bool
natCase (C:U) (a b: C): nat -> C
natRec  (C:U) (z: C) (s: nat->C->C) : (n:nat) -> C

natElim (C:nat->U) (z: C zero)
        (s: (n:nat)->C(succ n)): (n:nat) -> C(n)
natInd  (C:nat->U) (z: C zero)
        (s: (n:nat)->C(n)->C(succ n)): (n:nat) -> C(n)
```

3.2.8. List.

Визначення 64. (List). The data type of list L over a given set A can be represented as the initial algebra $(\mu L_A, \mathrm{in})$ of the functor $L_A(X) = 1 + (AX)$. Denote $\mu L_A = \mathrm{List}(A)$. The constructor functions $\mathrm{nil} : 1 \to \mathrm{List}(A)$ and $\mathrm{cons} : A \times \mathrm{List}(A) \to \mathrm{List}(A)$ are defined by $\mathrm{nil} = \mathrm{in} \circ \mathrm{inl}$ and $\mathrm{cons} = \mathrm{in} \circ \mathrm{inr}$, so $\mathrm{in} = [\mathrm{nil}, \mathrm{cons}]$.

```
data list (A: U) = nil | cons (x:A) (xs: list A)
listCase (A C:U) (a b: C): list A -> C
listRec (A C:U) (z: C) (s: A->list A->C->C): (n:list A) -> C
listElim (A: U) (C:list A->U) (z: C nil)
   (s: (x:A)(xs:list A)->C(cons x xs)): (n:list A) -> C(n)
listInd (A: U) (C:list A->U) (z: C nil)
   (s: (x:A)(xs:list A)->C(xs)->C(cons x xs)): (n:list A) -> C(n)

null (A:U): list A -> bool
head (A:U): list A -> maybe A
tail (A:U): list A -> maybe (list A)
nth (A:U): nat -> list A -> maybeA
append (A: U): list A -> list A -> list A
reverse (A: U): list A -> list A
map (A B: U): (A -> B) -> list A -> list B
zip (AB: U): list A -> list B -> list (tuple A B)
foldr (AB: U): (A -> B -> B) -> B -> list A -> B
foldl (AB: U): (B -> A -> B) -> B -> list A -> B
switch (A: U): (Unit -> list A) -> bool -> list A
filter (A: U): (A -> bool) -> list A -> list A
length (A: U): list A -> nat
listEq (A: eq): list A.1 -> list A.1 -> bool
```

3.2.9. Stream.

stream is a record form of the list's cons constructor. It models the infinity list that has no terminal element.

```
data stream (A: U) = cons (x: A) (xs: stream A)
```

3.2.10. Fin. fin is the inductive defintion of set with finite elements.

```
data fin (n: nat)
   = fzero | fsucc (_: fin (pred n))

fz (n: nat): fin (succ n)          = fzero
fs (n: nat): fin n -> fin (succ n) = \(x: fin n) -> fsucc x
```

3.2.11. Vector. vector is the inductive defintion of limited length list.

```
data vector (A: U) (n: nat)
   = nil | cons (_: A) (_: vector A (pred n))
```

seq — abstract compositional sequences.

```
data seq (A: U) (B: A -> A -> U) (X Y: A)
   = seqNil (_: A)
   | seqCons (X Y Z: A) (_: B X Y) (_: Seq A B Y Z)
```

3.2.12. Імпредикативне кодування. You know Church encoding which also has its dependent alanolgue in CoC, however in Coq it is imposible to detive Inductive Principle as type system lacks fixpoint and functional extensionality. The example of working compiler of PTS languages are Om and Morte. Assume we have Church encoded NAT:

```
nat = (X:U) -> (X -> X) -> X -> X
```

where first parameter $(X- > X)$ is a $succ$, the second parameter X is $zero$, and the result of encoding is landed in X. Even if we encode the parameter

```
list (A: U) = (X:U) -> X -> (A -> X) -> X
```

and paremeter A let's say live in 42 universe and X live in 2 universe, then by the signature of encoding the term will be landed in X, thus 2 universe. In other words such dependency is called impredicative displaying that landed term is not a predicate over parameters. This means that Church encoding is incompatible with predicative type checkers with predicative of predicative-cumulative hierarchies.

In HoTT n-types is encoded as n-groupoids, thus we need to add a predicate in which n-type we would like to land the encoding:

```
NAT (A: U) = (X:U) -> isSet X -> X -> (A -> X) -> X
```

Here we added isSet predicate. With this motto we can implement propositional truncation by landing term in isProp or even HIT by langing in isGroupoid:

```
TRUN (A:U) type = (X: U) -> isProp X -> (A -> X) -> X
S1 = (X:U) -> isGroupoid X -> ((x:X) -> Path X x x) -> X
MONOPLE (A:U) = (X:U) -> isSet X -> (A -> X) -> X
NAT = (X:U) -> isSet X -> X -> (A -> X) -> X
```

The main publication on this topic could be found at [14] and [15]. Here we have the implementation of Unit impredicative encoding in HoTT.

```
upPath      (X Y:U)(f:X->Y)(a:X->X): X -> Y = o X X Y f a
downPath    (X Y:U)(f:X->Y)(b:Y->Y): X -> Y = o X Y Y b f
naturality (X Y:U)(f:X->Y)(a:X->X)(b:Y->Y): U
  = Path (X->Y)(upPath X Y f a)(downPath X Y f b)

unitEnc': U = (X: U) -> isSet X -> X -> X
isUnitEnc (one: unitEnc'): U
  = (X Y:U)(x:isSet X)(y:isSet Y)(f:X->Y) ->
    naturality X Y f (one X x)(one Y y)

unitEnc: U = (x: unitEnc') * isUnitEnc x
unitEncStar: unitEnc = (\(X:U)(_:isSet X) ->
  idfun X,\(X Y: U)(_:isSet X)(_:isSet Y)->refl(X->Y))
```

```
unitEncRec  (C: U) (s: isSet C) (c: C): unitEnc -> C
  = \(z: unitEnc) -> z.1 C s c
unitEncBeta (C: U) (s: isSet C) (c: C)
  : Path C (unitEncRec C s c unitEncStar) c = refl C c
unitEncEta (z: unitEnc): Path unitEnc unitEncStar z = undefined
unitEncInd (P: unitEnc -> U) (a: unitEnc): P unitEncStar -> P a
  = subst unitEnc P unitEncStar a (unitEncEta a)
unitEncCondition (n: unitEnc'): isProp (isUnitEnc n)
  =  \(f g: isUnitEnc n) ->
     <h> \(x y: U) -> \(X: isSet x) -> \(Y: isSet y)
  -> \(F: x -> y) -> <i> \(R: x) -> Y (F (n x X R)) (n y Y (F R))
       (<j> f x y X Y F @ j R) (<j> g x y X Y F @ j R) @ h @ i
```

3.3. Гомотопічна теорія типів

Homotypy Type Theory takes its origins in 1996 from groupoid interpretation by Hofmann and Streicher's, and later (in 10 years) was formalized by Awodey, Warren and Voevodsky. Voevodsky constrtucted Kan simplicial sets interpretation of type theory and discovered the property of this model, that was named univalence. This property allows to identify isomorphic structures in terms of type theory.

Homotopy type theory to classical homotopy theory is like Euclidian syntethic geometry (points, lines, axioms and deduction rules) to analytical geometry with cartesian coordinates on $\mathbb{R}^n$ (geometric and algebraic) [11]

In the same way as inductive types extends MLTT for inductive programming, the higher inductive types (HIT) extend homotopy type

[11] We will denote geometric, type theoretical and homotopy constants bold font **R** while analitical will be denoted with double lined letters $\mathbb{R}$.

theory for geometry programming. You can directly encode CW-complexes by using HIT. The definition of HIT syntax will be given in the next **Issue IV: Higher Inductive Types**.

3.3.1. Гомотопії. The first higher equality we meet in homotopy theory is a notion of homotopy, where we compare two functions or two path spaces (which is sort of dependent families). The homotopy interval $I = [0, 1]$ is the perfect foundation for definition of homotopy.

Визначення 65. (Interval). Compact interval.

```
data I = i0
       | i1
       | seg <i> [(i=0) -> i0,
                  (i=1) -> i1]
```

You can think of **I** as isomorphism of equality type, disregarding carriers on the edges. By mapping $i0, i1 : \mathbf{I}$ to $x, y : A$ one can obtain identity or equality type from classic type theory.

Визначення 66. (Interval Split). The convertion function from I to a type of comparison is a direct eliminator of interval. The interval is also known as one of primitive higher inductive types which will be given in the next **Issue IV: Higher Inductive Types**.

```
pathToHtpy (A: U) (x y: A) (p: Path A x y): I -> A
   = split { i0 -> x; i1 -> y; seg @ i -> p @ i }
```

Визначення 67. (Homotopy). The homotopy between two function

$f, g : X \to Y$ is a continuous map of cylinder $H : X \times \mathbf{I} \to Y$ such that

$$\begin{cases} H(x, 0) = f(x), \\ H(x, 1) = g(x). \end{cases}$$

```
homotopy (X Y: U) (f g: X -> Y)
         (p: (x: X) -> Path Y (f x) (g x))
         (x: X): I -> Y = pathToHtpy Y (f x) (g x) (p x)
```

3.3.2. Групоїдна інтерпретація. The first text about groupoid interpretation of type theory can be found in Francois Lamarche: A proposal about Foundations[12]. Then Martin Hofmann and Thomas Streicher wrote the initial document on groupoid interpretation of type theory[13].

Equality	**Homotopy**	**∞-Groupoid**
reflexivity	constant path	identity morphism
symmetry	inversion of path	inverse morphism
transitivity	concatenation of paths	composition of mopphisms

There is a deep connection between higher-dimentinal groupoids in category theory and spaces in homotopy theory, equipped with some topology. The category or groupoid could be built where the objects are particular spaces or types, and morphisms are path types between these types, composition operation is a path concatenation. We can write this groupoid here recalling that it should be category with inverted morphisms.

[12] http://www.cse.chalmers.se/~coquand/Proposal.pdf

[13] Martin Hofmann and Thomas Streicher. The Groupoid Interpretation of Type Theory. 1996.

```
cat: U = (A: U) * (A -> A -> U)
groupoid: U = (X: cat) * isCatGroupoid X
PathCat (X: U): cat = (X,\(x y:X)->Path X x y)

isCatGroupoid (C: cat): U
  = (id: (x: C.1) -> C.2 x x)
  * (c: (x y z:C.1) -> C.2 x y -> C.2 y z -> C.2 x z)
  * (inv: (x y: C.1) -> C.2 x y -> C.2 y x)
  * (inv_left:  (x y: C.1) (p: C.2 x y) ->
    Path (C.2 x x) (c x y x p (inv x y p)) (id x))
  * (inv_right: (x y: C.1) (p: C.2 x y) ->
    Path (C.2 y y) (c y x y (inv x y p) p) (id y))
  * (left: (x y: C.1) (f: C.2 x y) ->
    Path (C.2 x y) (c x x y (id x) f) f)
  * (right: (x y: C.1) (f: C.2 x y) ->
    Path (C.2 x y) (c x y y f (id y)) f)
  * ((x y z w:C.1)(f:C.2 x y)(g:C.2 y z)(h:C.2 z w)->
    Path (C.2 x w) (c x z w (c x y z f g) h)
                   (c x y w f (c y z w g h)))

PathGrpd (X: U)
  : groupoid
  = ((Ob,Hom),id,c,sym X,compPathInv X,compInvPath X,L,R,Q) where
    Ob: U = X
    Hom (A B: Ob): U = Path X A B
    id (A: Ob): Path X A A = refl X A
    c (A B C: Ob) (f: Hom A B) (g: Hom B C): Hom A C
      = comp (<i> Path X A (g@i)) f []
```

From here should be clear what it meant to be groupoid interpretation of path type in type theory. In the same way we can construct categories of $\prod$ and $\sum$ types. In **Issue VIII: Topos Theory** such categories will be given.

3.3.3. Функціональна екстенсіональність.

Визначення 68. (funExt-Formation)

```
funext_form (A B: U) (f g: A -> B): U
  = Path (A -> B) f g
```

Визначення 69. (funExt-Introduction)

```
funext (A B: U) (f g: A -> B) (p: (x:A) -> Path B (f x) (g x))
  : funext_form A B f g
  = <i> \(a: A) -> p a @ i
```

Визначення 70. (funExt-Elimination)

```
happly (A B: U) (f g: A -> B) (p: funext_form A B f g) (x: A)
  : Path B (f x) (g x)
  = cong (A -> B) B (\(h: A -> B) -> apply A B h x) f g p
```

Визначення 71. (funExt-Computation)

```
funext_Beta (A B: U) (f g: A -> B) (p: (x:A) -> Path B (f x) (g x))
  : (x:A) -> Path B (f x) (g x)
  = \(x:A) -> happly A B f g (funext A B f g p) x
```

Визначення 72. (funExt-Uniqueness)

```
funext_Eta (A B: U) (f g: A -> B) (p: Path (A -> B) f g)
  : Path (Path (A -> B) f g) (funext A B f g (happly A B f g p)) p
  = refl (Path (A -> B) f g) p
```

3.3.4. Пулбеки.

Визначення 73. (Пулбек).

```
pullback (A B C:U) (f: A -> C) (g: B -> C): U
  = (a: A)
  * (b: B)
  * Path C (f a) (g b)
```

```
pb1 (A B C: U) (f: A -> C) (g: B -> C)
  : pullback A B C f g -> A
  = \(x: pullback A B C f g) -> x.1

pb2 (A B C: U) (f: A -> C) (g: B -> C)
  : pullback A B C f g -> B
  = \(x: pullback A B C f g) -> x.2.1

pb3 (A B C: U) (f: A -> C) (g: B -> C)
  : (x: pullback A B C f g) -> Path C (f x.1) (g x.2.1)
  = \(x: pullback A B C f g) -> x.2.2
```

Визначення 74. (Ядро).

```
kernel  (A B: U) (f: A -> B): U
  = pullback A A B f f
```

Визначення 75. (Гомотопічне розшарування).

```
hofiber (A B: U) (f: A -> B) (y: B): U
  = pullback A unit B f (\(x: unit) -> y)
```

Визначення 76. (Пулбек Квадрат).

```
pullbackSq (Z A B C: U) (f: A -> C) (g: B -> C) (z1: Z -> A) (z2: Z -> B): U
         = (h: (z:Z) -> Path C ((o Z A C f z1) z) (((o Z B C g z2)) z))
         * isEquiv Z (pullback A B C f g) (induced Z A B C f g z1 z2 h)
```

Теорема 24. (Існування пулбеку).

```
completePullback (A B C: U) (f: A -> C) (g: B -> C)
    : pullbackSq (pullback A B C f g) A B C f g (pb1 A B C f g) (pb2 A B C f g)
```

3.3.5. Фібрації.

Визначення 77. (Fibration-1) Dependent fiber bundle derived from Path contractability.

```
isFBundle1 (B: U) (p: B -> U) (F: U): U
  = (_: (b: B) -> isContr (Path U (p b) F))
  * ((x: Sigma B p) -> B)
```

Визначення 78. (Fibration-2). Dependent fiber bundle derived from surjective function.

```
isFBundle2 (B: U) (p: B -> U) (F: U): U
  = (V: U)
  * (v: surjective V B)
  * ((x: V) -> Path U (p (v.1 x)) F)
```

Визначення 79. (Fibration-3). Non-dependent fiber bundle derived from fiber truncation.

```
im1 (A B: U) (f: A -> B): U = (b: B) * pTrunc ((a:A) * Path B (f a) b)
BAut (F: U): U = im1 unit U (\(x: unit) -> F)
unitIm1 (A B: U) (f: A -> B): im1 A B f -> B = \(x: im1 A B f) -> x.1
unitBAut (F: U): BAut F -> U = unitIm1 unit U (\(x: unit) -> F)

isFBundle3 (E B: U) (p: E -> B) (F: U): U
  = (X: B -> BAut F)
  * (classify B (BAut F) (\(b: B) -> fiber E B p b) (unitBAut F) X) where
  classify (A' A: U) (E': A' -> U) (E: A -> U) (f: A' -> A): U
    = (x: A') -> Path U (E'(x)) (E(f(x)))
```

Визначення 80. (Fibration-4). Non-dependen fiber bundle derived as pullback square.

```
isFBundle4 (E B: U) (p: E -> B) (F: U): U
  = (V: U)
  * (v: surjective V B)
  * (v': prod V F -> E)
  * pullbackSq (prod V F) E V B p v.1 v' (\(x: prod V F) -> x.1)
```

3.3.6. Еквівалентність.

Визначення 81. (Equivalence).

```
fiber (A B: U) (f: A -> B) (y: B): U = (x: A) * Path B y (f x)
isSingleton (X:U): U = (c:X) * ((x:X) -> Path X c x)
isEquiv (A B: U) (f: A -> B): U = (y: B) -> isContr (fiber A B f y)
equiv (A B: U): U = (f: A -> B) * isEquiv A B f
```

Визначення 82. (Surjective).

```
isSurjective (A B: U) (f: A -> B): U
  = (b: B) * pTrunc (fiber A B f b)

surjective (A B: U): U
  = (f: A -> B)
  * isSurjective A B f
```

Визначення 83. (Injective).

```
isInjective' (A B: U) (f: A -> B): U
  = (b: B) -> isProp (fiber A B f b)

injective (A B: U): U
  = (f: A -> B)
  * isInjective A B f
```

Визначення 84. (Embedding).

```
isEmbedding (A B: U) (f: A -> B) : U
  = (x y: A) -> isEquiv (Path A x y) (Path B (f x) (f y)) (cong A B f x y)

embedding (A B: U): U
  = (f: A -> B)
  * isEmbedding A B f
```

Визначення 85. (Half-adjoint Equivalence).

```
isHae (A B: U) (f: A -> B): U
  = (g: B -> A)
  * (eta_: Path (id A) (o A B A g f) (idfun A))
  * (eps_: Path (id B) (o B A B f g) (idfun B))
  * ((x: A) -> Path B (f ((eta_ @ 0) x)) ((eps_ @ 0) (f x)))

hae (A B: U): U
  = (f: A -> B)
  * isHae A B f
```

3.3.7. Ізоморфізм.

Визначення 86. (iso-Formation)

```
iso_Form (A B: U): U = isIso A B -> Path U A B
```

Визначення 87. (iso-Introduction)

```
iso_Intro (A B: U): iso_Form A B
```

Визначення 88. (iso-Elimination)

```
iso_Elim (A B: U): Path U A B -> isIso A B
```

Визначення 89. (iso-Computation)

```
iso_Comp (A B : U) (p : Path U A B)
  : Path (Path U A B) (iso_Intro A B (iso_Elim A B p)) p
```

Визначення 90. (iso-Uniqueness)

```
iso_Uniq (A B : U) (p: isIso A B)
  : Path (isIso A B) (iso_Elim A B (iso_Intro A B p)) p
```

3.3.8. Унівалентність.

Визначення 91. (uni-Formation)

```
univ_Formation (A B: U): U = equiv A B -> Path U A B
```

Визначення 92. (uni-Introduction)

```
equivToPath (A B: U): univ_Formation A B
  = \(p: equiv A B) -> <i> Glue B [(i=0) -> (A,p),
    (i=1) -> (B, subst U (equiv B) B B (<_>B) (idEquiv B)) ]
```

Визначення 93. (uni-Elimination)

```
pathToEquiv (A B: U) (p: Path U A B) : equiv A B
  = subst U (equiv A) A B p (idEquiv A)
```

Визначення 94. (uni-Computation)

```
eqToEq (A B : U) (p : Path U A B)
  : Path (Path U A B) (equivToPath A B (pathToEquiv A B p)) p
  = <j i> let Ai: U = p@i in Glue B
    [ (i=0) -> (A,pathToEquiv A B p),
      (i=1) -> (B,pathToEquiv B B (<k> B)),
      (j=1) -> (p@i,pathToEquiv Ai B (<k> p @ (i \/ k))) ]
```

Визначення 95. (uni-Uniqueness)

```
transPathFun (A B : U) (w: equiv A B)
  : Path (A -> B) w.1 (pathToEquiv A B (equivToPath A B w)).1
```

3.4. Вищі індуктивні типи

CW-complexes are fundamental objects in homotopy type theory and even included inside cubical type checker in a form of higher (co)-inductive types (HITs). Just like regular (co)-inductive types could be described as recur- sive terminating (well-founded) or non-terminating trees, higher inductive types could be described as CW-complexes. Defining HIT means to define some CW- complex directly using cubical homogeneous composition structure as an element of initial algebra inside cubical model.

3.4.1. Інтервал.

Визначення 96. (Interval). Compact interval.

```
data I = i0
       | i1
       | seg <i> [(i=0) -> i0,
                  (i=1) -> i1]
```

You can think of **I** as isomorphism of equality type, disregarding carriers on the edges. By mapping $i0, i1 : \mathbf{I}$ to $x, y : A$ one can obtain identity or equality type from classic type theory.

3.4.2. n-Сфера.

Визначення 97. (Shperes and Disks). Here are some example of using dimensions to construct spherical shapes.

```
data S1
   = base
   | loop <i> [ (i = 0) -> base,
                (i = 1) -> base ]

data S2
   = point
   | surf <i j> [ (i = 0) -> point, (i = 1) -> point,
                  (j = 0) -> point, (j = 1) -> point ]
                  (j = 0) -> point, (j = 1) -> point ]
```

3.4.3. Суспензія.

Визначення 98. (Suspension).

```
data susp (A: U)
  = north
  | south
  | merid (a: A) <i> [ (i = 0) -> north ,
                       (i = 1) -> south ]
```

3.4.4. Транкейшин.

Визначення 99. (Truncation).

```
data pTrunc (A: U) -- (-1)-trunc, mere proposition truncation
  = pinc (a: A)
  | pline (x y: pTrunc A) <i>
        [ (i = 0) -> x,
          (i = 1) -> y ]

data sTrunc (A: U) -- (0)-trunc, set truncation
  = sinc (a: A)
  | sline (a b: sTrunc A)
          (p q: Path (sTrunc A) a b) &lt;i j&gt;
        [ (i = 0) -> p @ j,
          (i = 1) -> q @ j,
          (j = 0) -> a,
          (j = 1) -> b ]

data gTrunc (A: U) -- (1)-trunc, groupoid truncation
  = ginc   (a: A)
  | gline  (a b: gTrunc A)
           (p q: Path (gTrunc A) a b)
           (r s: Path (Path (gTrunc A) a b) p q) &lt;i j k&gt;
         [ (i = 0) -> r @ j @ k,
           (i = 1) -> s @ j @ k,
           (j = 0) -> p @ k,
           (j = 1) -> q @ k,
           (k = 0) -> a,
           (k = 1) -> b ]
```

3.4.5. Факторизація.

Визначення 100. (Quotient).

```
data quot (A: U) (R: A -> A -> U)
  = inj (a: A)
  | quoteq (a b: A) (r: R a b) &lt;i&gt;
         [ (i = 0) -> inj a,
           (i = 1) -> inj b ]


data setquot (A: U) (R: A -> A -> U)
  = quotient (a: A)
  | identification (a b: A) (r: R a b) &lt;i&gt;
                 [ (i = 0) -> quotient a,
                   (i = 1) -> quotient b ]
  | setTruncation  (a b: setquot A R)
                   (p q: Path (setquot A R) a b) &lt;i j&gt;
                 [ (i = 0) -> p @ j,
                   (i = 1) -> q @ j,
                   (j = 0) -> a,
                   (j = 1) -> b ]
```

3.4.6. Пушаут.

Визначення 101. (Pushout). One of the notable examples is pushout as it's used to define the cell attachment formally, as others cofibrant objects.

```
data pushout (A B C: U) (f: C -> A) (g: C -> B)
   = po1 (_: A)
   | po2 (_: B)
   | po3 (c: C) <i> [ (i = 0) -> po1 (f c) ,
                      (i = 1) -> po2 (g c) ]
```

3.5. Модальності

3.5.1. Процеси. Process Calculus defines formal business process engine that could be mapped onto Synrc/BPE Erlang/OTP application or OCaml Lwt library with Coq.io front-end. Here we will describe an Erlang approach for modeling processes. We will describe process calculus as a formal model of two types: 1) the general abstract MLTT interface of process modality that can be used as a formal binding to low-level programming or as a top-level interface; 2) the low-level formal model of Erlang/OTP generic server.

Визначення 102. (Storage). The secure storage based on verified cryptography. NOTE: For simplicity let it be a compatible list.

```
storage: U -> U = list
```

Визначення 103. (Process). The type formation rule of the process is a Σ telescope that contains: i) protocol type; ii) state type; iii) in-memory current state of process in the form of cartesian product of protocol and state which is called signature of the process; iv) monoidal action on signature; v) persistent storage for process trace.

```
process : U
  = (protocol state: U)
  * (current: prod protocol state)
  * (act: id (prod protocol state))
  * (storage (prod protocol state))
```

Визначення 104. (Spawn). The sole introduction rule, process constructor is a tuple with filled process type information. Spawn is a modal arrow representing the fact that process instance is created at some scheduler of CPU core.

```
spawn (protocol state: U) (init: prod protocol state)
      (action: id (prod protocol state)) : process
    = (protocol,state,init,action,nil)
```

Визначення 105. (Accessors). Process type defines following accessors (projections, this eliminators) to its structure: i) protocol type; ii) state type; iii) signature of the process; iv) current state of the process; v) action projection; vi) trace projection.

```
protocol  (p: process): U = p.1
state     (p: process): U = p.2.1
signature (p: process): U = prod p.1 p.2.1
current   (p: process):          signature p  = p.2.2.1
action    (p: process):       id (signature p) = p.2.2.2.1
trace     (p: process): storage (signature p) = p.2.2.2.2
```

NOTE: there are two kinds of approaches to process design: 1) Semigroup: $P \times S \rightarrow S$; and 2) Monoidal: $P \times S \rightarrow P \times S$, where P is protocol and S is state of the process.

Визначення 106. (Receive). The modal arrow that represents sleep of the process until protocol message arrived.

```
receive (p: process) : protocol p = axiom
```

Визначення 107. (Send). The response free function that represents sending a message to a particular process in the run-time. The Send nature is async and invisible but is a part of process modality as it’s effectfull.

```
send (p: process) (message: protocol p) : unit = axiom
```

Визначення 108. (Execute). The Execute function is an eliminator of process stream performing addition of a single entry to the secured storage of the process. Execute is a transactional or synchronized version of asynchronous Send.

```
execute (p: process) (message: protocol p) : process
  = let step: signature p = (action p) (message, (current p).2)
     in (protocol p, state p, step, action p, cons step (trace p))
```

1) Run-time formal model of Erlang/OTP compatible generic server with extraction to Erlang. This is an example of low-level process modality usage. The run-time formal model can be seen here[14].

2) Formal model of Business Process Engine application that runs on top of Erlang/OTP extracted model. The Synrc/BPE model can be seen here[15].

3) Formal model of Synrc/N2O application and n2o_async[16] in particular.

[14] https://n2o.space/articles/streams.htm

[15] https://n2o.space/articles/bpe.htm

[16] https://mqtt.n2o.space/man/n2o_async.htm

РОЗДІЛ 4
МАТЕМАТИЧНІ КОМПОНЕНТИ

Четвертий розділ надає приклади математичного моделювання та складних теорем теорії категорій, теорій топосів, теорії гомотопій, тощо.

4.1. Теорія категорій

4.1.1. Категорія.

Визначення 109. (Category Signature). The signature of category is a $\Sigma_{A:U} A \to A \to U$ where U could be any universe. The pr_1 projection is called Ob and pr_2 projection is called $Hom(a, b)$, where $a, b : Ob$.

```
cat: U = (A: U) * (A -> A -> U)
```

Precategory C defined as set of $Hom_C(a, b)$ where $a, b : Ob_C$ are objects defined by its id arrows $Hom_C(x, x)$. Properfies of left and right units included with composition c and its associativity.

Визначення 110. (Precategory). More formal, precategory C consists of the following. (i) A type Ob_C, whose elements are called objects; (ii) for each $a, b : Ob_C$, a set $Hom_C(a, b)$, whose elements are called arrows or morphisms. (iii) For each $a : Ob_C$, a morphism $1_a : Hom_C(a, a)$, called the identity morphism. (iv) For each $a, b, c : Ob_C$, a function $Hom_C(b, c) \to$

$\mathrm{Hom}_C(a,b) \to \mathrm{Hom}_C(a,c)$ called composition, and denoted $g \circ f$. (v) For each $a, b : \mathrm{Ob}_C$ and $f : \mathrm{Hom}_C(a,b)$, $f = 1_b \circ f$ and $f = f \circ 1_a$. (vi) For each $a, b, c, d : A$ and $f : \mathrm{Hom}_C(a,b)$, $g : \mathrm{Hom}_C(b,c)$, $h : \mathrm{Hom}_C(c,d)$, $h \circ (g \circ f) = (h \circ g) \circ f$.

Визначення 111. (Small Category). If for all $a, b : \mathrm{Ob}$ the $\mathrm{Hom}_C(a,b)$ forms a Set, then such category is called small category.

```
isPrecategory (C: cat): U
  = (id: (x: C.1) -> C.2 x x)
  * (c: (x y z: C.1) -> C.2 x y -> C.2 y z -> C.2 x z)
  * (homSet: (x y: C.1) -> isSet (C.2 x y))
  * (left: (x y: C.1) -> (f: C.2 x y)
  -> Path (C.2 x y) (c x x y (id x) f) f)
  * (right: (x y: C.1) -> (f: C.2 x y)
  -> Path (C.2 x y) (c x y y f (id y)) f)
  * ( (x y z w: C.1) (f: C.2 x y) (g: C.2 y z)
    (h: C.2 z w) -> Path (C.2 x w)
    (c x z w (c x y z f g) h) (c x y w f (c y z w g h)))
```

precategory: U = (C: cat) * isPrecategory C

Accessors of the precategory structure. For Ob is carrier and for Hom is hom.

```
carrier (C: precategory): U = C.1.1
hom     (C: precategory) (a b: carrier C): U = C.1.2 a b
path    (C: precategory) (x: carrier C): hom C x x = C.2.1 x
compose (C: precategory) (x y z: carrier C)
        (f: hom C x y) (g: hom C y z): hom C x z = C.2.2.1 x y z f g
```

4.1.2. (Ко)термінал.

Визначення 112. (Initial Object). Is such object Ob_C, that $\Pi_{x,y:\mathrm{Ob}_C}$ $\mathrm{isContr}(\mathrm{Hom}_C(x,y))$.

Визначення 113. (Terminal Object). Is such object Ob_C, that $\Pi_{x,y:\mathrm{Ob}_C}$ $\mathrm{isContr}(\mathrm{Hom}_C(y,x))$.

```
isInitial (C: precategory) (x: carrier C): U
  = (y: carrier C) -> isContr (hom C x y)
isTerminal (C: precategory) (y: carrier C): U
  = (x: carrier C) -> isContr (hom C x y)
initial (C: precategory): U
  = (x: carrier C) * isInitial  C x
terminal(C: precategory): U
  = (y: carrier C) * isTerminal C y
```

4.1.3. Функтор.

Визначення 114. (Category Functor). Let A and B be precategories. A functor $F : A \to B$ consists of: (i) A function $F_{\mathrm{Ob}} : \mathrm{Ob}_h A \to \mathrm{Ob}_B$; (ii) for each $a,b : \mathrm{Ob}_A$, a function $F_{\mathrm{Hom}} : \mathrm{Hom}_A(a,b) \to \mathrm{Hom}_B(F_{\mathrm{Ob}}(a), F_{\mathrm{Ob}}(b))$; (iii) for each $a : \mathrm{Ob}_A$, $F_{\mathrm{Ob}}(1_a) = 1_{F_{\mathrm{Ob}}}(a)$; (iv) for $a,b,c : \mathrm{Ob}_A$ and $f : \mathrm{Hom}_A(a,b)$ and $g : \mathrm{Hom}_A(b,c)$, $F(g \circ f) = F_{\mathrm{Hom}}(g) \circ F_{\mathrm{Hom}}(f)$.

```
catfunctor (A B: precategory): U
  = (ob: carrier A -> carrier B)
  * (mor: (x y: carrier A) -> hom A x y -> hom B (ob x) (ob y))
  * (id: (x: carrier A) -> Path (hom B (ob x) (ob x))
    (mor x x (path A x)) (path B (ob x)))
  * ((x y z: carrier A) -> (f: hom A x y) -> (g: hom A y z) ->
     Path (hom B (ob x) (ob z)) (mor x z (compose A x y z f g))
      (compose B (ob x) (ob y) (ob z) (mor x y f) (mor y z g)))
```

4.1.4. Натуральні перетворення.

Визначення 115. (Natural Transformation). For functors $F, G : C \to D$, a nagtural transformation $\gamma : F \to G$ consists of: (i) for each $x : C$, a morphism $\gamma_a : \mathrm{Hom}_D(F(x), G(x))$; (ii) for each $x, y : C$ and $f : \mathrm{Hom}_C(x, y)$, $G(f) \circ \gamma_x = \gamma_y \circ F(g)$.

```
isNaturalTrans (C D: precategory)
               (F G: catfunctor C D)
               (eta: (x: carrier C) -> hom D (F.1 x) (G.1 x)): U
  = (x y: carrier C) (h: hom C x y) ->
     Path (hom D (F.1 x) (G.1 y))
     (compose D (F.1 x) (F.1 y) (G.1 y) (F.2.1 x y h) (eta y))
     (compose D (F.1 x) (G.1 x) (G.1 y) (eta x) (G.2.1 x y h))

ntrans (C D: precategory) (F G: catfunctor C D): U
  = (eta: (x: carrier C) -> hom D (F.1 x) (G.1 x))
  * (isNaturalTrans C D F G eta)
```

4.1.5. Розширення Кана.

Визначення 116. (Kan Extension).

```
extension (C C' D: precategory)
    (K: catfunctor C C') (G: catfunctor C D) : U
  = (F: catfunctor C' D)
  * (ntrans C D (compFunctor C C' D K F) G)
```

4.1.6. Ізоморфізм категорій.

Визначення 117. (Category Isomorphism). A morphism $f : \mathrm{Hom}_A(a, b)$ is an iso if there is a morphism $g : \mathrm{Hom}_A(b, a)$ such that $1_a =_\eta g \circ f$ and $f \circ g =_\epsilon 1_b = g$. "f is iso"is a mere proposition. If A is a precategory and $a, b : A$, then $a = b \to \mathrm{iso}_A(a, b)$ (idtoiso).

```
iso (C: precategory) (A B: carrier C): U
  = (f: hom C A B)
  * (g: hom C B A)
  * (eta: Path (hom C A A) (compose C A B A f g) (path C A))
  * (Path (hom C B B) (compose C B A B g f) (path C B))
```

4.1.7. Резк-поповнення.

Визначення 118. (Category). A category is a precategory such that for all $a : \mathrm{Ob}_C$, the $\Pi_{A:\mathrm{Ob}_C} \mathrm{isContr} \Sigma_{B:\mathrm{Ob}_C} \mathrm{iso}_C(A, B)$.

```
isCategory (C: precategory): U
  = (A: carrier C) -> isContr ((B: carrier C) * iso C A B)
    category: U = (C: precategory) * isCategory C
```

4.1.8. Конструкції.

4.1.8.1. (Ко)продукти категорій.

Визначення 119. (Category Product).

```
Product     (X Y: precategory) : precategory
Coproduct   (X Y: precategory) : precategory
```

4.1.8.2. Обернена категорія.

Визначення 120. (Opposite Category). The opposite category to category C is a category C^{op} with same structure, except all arrows are inverted.

```
opCat (P: precategory): precategory
```

4.1.8.3. (Ко)слайс категорія.

Визначення 121. (Slice Category).

Визначення 122. (Coslice Category).

```
sliceCat (C D: precategory)
    (a: carrier (opCat C))
    (F: catfunctor D (opCat C))
  : precategory
  = cosliceCat (opCat C) D a F

cosliceCat (C D: precategory)
  (a: carrier C)
  (F: catfunctor D C) : precategory
```

4.1.8.4. Універсальна властивість.

Визначення 123. (Universal Mapping Property).

```
initArr (C D: precategory)
  (a: carrier C)
  (F: catfunctor D C): U = initial (cosliceCat C D a F)

termArr (C D: precategory)
  (a: carrier (opCat C))
  (F: catfunctor D (opCat C)): U = terminal (sliceCat C D a F)
```

4.1.8.5. Одинична категорія.

Визначення 124. (Unit Category). In unit category both $\mathrm{Ob} = \top$ and $\mathrm{Hom} = \top$.

```
unitCat: precategory
```

4.1.9. Приклади.

4.1.9.1. Категорія множин.

Визначення 125. (Category of Sets).

```
Set: precategory = ((Ob,Hom),id,c,HomSet,L,R,Q) where
  Ob: U = SET
  Hom (A B: Ob): U = A.1 -> B.1
  id (A: Ob): Hom A A = idfun A.1
  c (A B C: Ob) (f: Hom A B) (g: Hom B C): Hom A C
   = o A.1 B.1 C.1 g f
  HomSet (A B: Ob): isSet (Hom A B) = setFun A.1 B.1 B.2
  L (A B: Ob) (f: Hom A B): Path (Hom A B) (c A A B (id A) f) f
   = refl (Hom A B) f
  R (A B: Ob) (f: Hom A B): Path (Hom A B) (c A B B f (id B)) f
   = refl (Hom A B) f
  Q (A B C D: Ob) (f: Hom A B) (g: Hom B C) (h: Hom C D)
   : Path (Hom A D) (c A C D (c A B C f g) h) (c A B D f (c B C D g h))
   = refl (Hom A D) (c A B D f (c B C D g h))
```

4.1.9.2. Категорія функцій.

Визначення 126. (Category of Functions over Sets).

```
Functions (X Y: U) (Z: isSet Y): precategory
   = ((Ob,Hom),id,c,HomSet,L,R,Q) where
  Ob: U = X -> Y
  Hom (A B: Ob): U = id (X -> Y)
  id (A: Ob): Hom A A = idfun (X -> Y)
  c (A B C: Ob) (f: Hom A B) (g: Hom B C): Hom A C = idfun (X -> Y)
  HomSet (A B: Ob): isSet (Hom A B) = setFun Ob Ob (setFun X Y Z)
  L (A B: Ob) (f: Hom A B): Path (Hom A B) (c A A B (id A) f) f = axiom
  R (A B: Ob) (f: Hom A B): Path (Hom A B) (c A B B f (id B)) f = axiom
  Q (A B C D: Ob) (f: Hom A B) (g: Hom B C) (h: Hom C D)
   : Path (Hom A D) (c A C D (c A B C f g) h)
                    (c A B D f (c B C D g h)) = axiom
```

4.1.9.3. Категорія категорій.

Визначення 127. (Category of Categories).

```
Cat: precategory = ((Ob,Hom),id,c,HomSet,L,R,Q) where
  Ob: U = precategory
  Hom (A B: Ob): U = catfunctor A B
  id (A: Ob): catfunctor A A = idFunctor A
  c (A B C: Ob) (f: Hom A B) (g: Hom B C): Hom A C
   = compFunctor A B C f g
  HomSet (A B: Ob): isSet (Hom A B) = axiom
  L (A B: Ob) (f: Hom A B): Path (Hom A B) (c A A B (id A) f) f = axiom
  R (A B: Ob) (f: Hom A B): Path (Hom A B) (c A B B f (id B)) f = axiom
  Q (A B C D: Ob) (f: Hom A B) (g: Hom B C) (h: Hom C D)
   : Path (Hom A D) (c A C D (c A B C f g) h)
                    (c A B D f (c B C D g h)) = axiom
```

4.1.9.4. Категорія функторів.

Визначення 128. (Category of Functors).

```
Func (X Y: precategory): precategory
   = ((Ob,Hom),id,c,HomSet,L,R,Q) where
  Ob: U = catfunctor X Y
  Hom (A B: Ob): U = ntrans X Y A B
  id (A: Ob): ntrans X Y A A = axiom
  c (A B C: Ob) (f: Hom A B) (g: Hom B C): Hom A C = axiom
  HomSet (A B: Ob): isSet (Hom A B) = axiom
  L (A B: Ob) (f: Hom A B): Path (Hom A B) (c A A B (id A) f) f = axiom
  R (A B: Ob) (f: Hom A B): Path (Hom A B) (c A B B f (id B)) f = axiom
  Q (A B C D: Ob) (f: Hom A B) (g: Hom B C) (h: Hom C D)
   : Path (Hom A D) (c A C D (c A B C f g) h)
                    (c A B D f (c B C D g h)) = axiom
```

4.1.10. k-морфізми.

Визначення 129. (k-Morphism). The k-morphism is defined as morphism between $(k-1)$-morphism. The base of induction, the 0-morphism is defined as object of 1-category, which is precategory.

```
equiv: U
functor (C D: cat): U
ntrans (C D: cat) (F G: functor C D): U
modification (C D: cat) (F G: functor C D) (I J: ntrans C D F G): U
```

І так далі.

4.1.11. 2-категорія.

Визначення 130. (2-Category).

```
Cat2 : U
  = (Ob: U)
  * (Hom:  (A B: Ob) -> U)
  * (Hom2: (A B: Ob) -> (C F: Hom A B) -> U)
  * (id:      (A: Ob) -> Hom A A)
  * (id2:     (A: Ob) -> (B: Hom A A) -> Hom2 A A B B)
  * (c:   (A B C: Ob) (f: Hom A B) (g: Hom B C) -> Hom A C)
  * (c2:    (A B: Ob) (X Y Z: Hom A B)
    (f: Hom2 A B X Y) (g: Hom2 A B Y Z) -> Hom2 A B X Z)
```

4.1.12. Аддитивна категорія.

4.1.13. Група Гротендіка.

4.1.14. Категорія Гротендіка.

4.2. Теорія топосів

One can admit two topos theory lineages. One lineage takes its roots from published by Jean Leray in 1945 initial work on sheaves and spectral sequences. Later this lineage was developed by Henri Paul Cartan, André Weil. The peak of lineage was settled with works by Jean-Pierre Serre, Alexander Grothendieck, and Roger Godement.

Second remarkable lineage take its root from William Lawvere and Myles Tierney. The main contribution is the reformulation of Grothendieck topology by using subobject classifier.

Визначення 131. (Categorical Pullback). The pullback of the cospan $A \xrightarrow{f} C \xleftarrow{g} B$ is a object $A \times_C B$ with morphisms $pb_1 : \times_C \to A$, $pb_2 : \times_C \to B$, such that diagram commutes:

$$\begin{tikzcd}
A \times_C B \arrow[r, "pb_2"] \arrow[d, "f"'] \arrow[dr] & B \arrow[d, "pb_1"] \\
A \arrow[r, "g"] & C
\end{tikzcd}$$

Pullback $(\times_C, pb_1, pb_2)$ must be universal, means for any (D, q_1, q_2) for which diagram also commutes there must exists a unique $u : D \to \times_C$, such that $pb_1 \circ u = q_1$ and $pb_2 \circ q_2$.

```
homTo  (C: precategory) (X: carrier C): U
  = (Y: carrier C) * hom C Y X
cospan (C: precategory): U
  = (X: carrier C) * (_: homTo C X) * homTo C X
cospanCone (C: precategory) (D: cospan C): U
  = (W: carrier C) * hasCospanCone C D W
```

```
cospanConeHom (C: precategory) (D: cospan C)
    (E1 E2: cospanCone C D) : U
  = (h: hom C E1.1 E2.1) * isCospanConeHom C D E1 E2 h
isPullback (C: precategory) (D: cospan C) (E: cospanCone C D) : U
  = (h: cospanCone C D) -> isContr (cospanConeHom C D h E)
hasPullback (C: precategory) (D: cospan C) : U
  = (E: cospanCone C D) * isPullback C D E
```

Визначення 132. (Category Functor). Let A and B be precategories. A functor $F : A \to B$ consists of: (i) A function $F_{Ob} : Ob_h A \to Ob_B$; (ii) for each $a, b : Ob_A$, a function $F_{Hom} : Hom_A(a, b) \to Hom_B(F_{Ob}(a), F_{Ob}(b))$; (iii) for each $a : Ob_A$, $F_{Ob}(1_a) = 1_{F_{Ob}}(a)$; (iv) for $a, b, c : Ob_A$ and $f : Hom_A(a, b)$ and $g : Hom_A(b, c)$, $F(g \circ f) = F_{Hom}(g) \circ F_{Hom}(f)$.

```
catfunctor (A B: precategory): U
  = (ob: carrier A -> carrier B)
  * (mor: (x y:carrier A)->hom A x y->hom B(ob x)(ob y))
  * (id: (x: carrier A) -> Path (hom B (ob x) (ob x))
    (mor x x (path A x)) (path B (ob x)))
  * ((x y z: carrier A) -> (f: hom A x y) -> (g: hom A y z) ->
    Path (hom B (ob x) (ob z)) (mor x z (compose A x y z f g))
         (compose B (ob x) (ob y) (ob z) (mor x y f) (mor y z g)))
```

Визначення 133. (Terminal Object). Is such object Ob_C, that

$$\prod_{x,y:Ob_C} \text{isContr}(\text{Hom}_C(y, x)).$$

```
isTerminal (C: precategory) (y: carrier C): U
  = (x: carrier C) -> isContr (hom C x y)
terminal (C: precategory): U
  = (y: carrier C) * isTerminal C y
```

4.2.1. Теорія множин. Here is given the ∞-groupoid model of sets.

Визначення 134. (Mere proposition, PROP). A type P is a mere proposition if for all $x, y : P$ we have $x = y$:

$$\mathrm{isProp}(P) = \prod_{x,y:P} (x = y).$$

Визначення 135. (0-type). A type A is a 0-type is for all $x, y : A$ and $p, q : x =_A y$ we have $p = q$.

Визначення 136. (1-type). A type A is a 1-type if for all $x, y : A$ and $p, q : x =_A y$ and $r, s : p =_{=_A} q$, we have $r = s$.

Визначення 137. (A set of elements, SET). A type A is a SET if for all $x, y : A$ and $p, q : x = y$, we have $p = q$:

$$\mathrm{isSet}(A) = \prod_{x,y:A} \prod_{p,q:x=y} (p = q).$$

Визначення 138. `data N = Z  | S (n: N)`

```
n_grpd (A: U) (n: N): U = (a b: A) -> rec A a b n where
  rec (A: U) (a b: A) : (k: N) -> U
    = split { Z -> Path A a b ; S n -> n_grpd (Path A a b) n }

isContr (A: U): U = (x: A) * ((y: A) -> Path A x y)
isProp  (A: U): U = n_grpd A Z
isSet   (A: U): U = n_grpd A (S Z)
PROP   : U = (X:U) * isProp X
SET    : U = (X:U) * isSet X
```

Визначення 139. (Π-Contractability). If fiber is set thene path space between any sections is contractible.

```
setPi (A: U) (B: A -> U) (h: (x: A) -> isSet (B x)) (f g: Pi A B)
      (p q: Path (Pi A B) f g)
    : Path (Path (Pi A B) f g) p q
```

Визначення 140. (Σ-Contractability). If fiber is set then Σ is set.

```
setSig (A:U) (B: A -> U) (base: isSet A)
       (fiber: (x:A) -> isSet (B x)) : isSet (Sigma A B)
```

Визначення 141. (Unit type, 1). The unit 1 is a type with one element.

```
data unit = tt
unitRec (C: U) (x: C): unit -> C = split tt -> x
unitInd (C: unit -> U) (x: C tt): (z:unit) -> C z
  = split tt -> x
```

Теорема 25. (Category of Sets, $\mathbf{Set}$). Sets forms a Category. All compositional theorems proved by using reflection rule of internal language. The proof that Hom forms a set is taken through Π-contractability.

```
Set: precategory = ((Ob,Hom),id,c,HomSet,L,R,Q) where
  Ob: U = SET
  Hom (A B: Ob): U = A.1 -> B.1
  id (A: Ob): Hom A A = idfun A.1
  c (A B C: Ob) (f: Hom A B) (g: Hom B C): Hom A C
    = o A.1 B.1 C.1 g f
  HomSet (A B: Ob): isSet (Hom A B) = setFun A.1 B.1 B.2
  L (A B:Ob) (f:Hom A B): Path (Hom A B)(c A A B (id A)f)f
    = refl (Hom A B) f
  R (A B:Ob) (f:Hom A B): Path (Hom A B)(c A B B f(id B))f
    = refl (Hom A B) f
  Q (A B C D: Ob) (f:Hom A B) (g:Hom B C) (h:Hom C D)
    : Path (Hom A D) (c A C D (c A B C f g) h)
                     (c A B D f (c B C D g h))
    = refl (Hom A D) (c A B D f (c B C D g h))
```

Topos theory extends category theory with notion of topological structure but reformulated in a categorical way as a category of sheaves on a site or as one that has cartesian closure and subobject classifier. We give here two definitions.

4.2.2. Топологічна структура.

Визначення 142. (Topology). The topological structure on A (or topology) is a subset $S \in A$ with following properties: i) any finite union of subsets of S is belong to S; ii) any finite intersection of subsets of S is belong to S. Subets of S are called open sets of family S.

```
Structure topology (A : Type) := {
          open :> (A -> Prop) -> Prop;
          empty_open: open (empty _);
          full_open:  open (full _);
          inter_open: forall u,
                      open u -> forall v, open v
                             -> open (inter A u v) ;
          union_open: forall s, (subset _ s open)
                             -> open (union A s) }.
```

For fully functional general topology theorems and Zorn lemma you can refer to the Coq library [1]topology by Daniel Schepler.

4.2.3. Топос Гротендіка. Grothendieck Topology is a calculus of coverings which generalizes the algebra of open covers of a topological space, and can exist on much more general categories. There are three variants of Grothendieck topology definition: i) sieves; ii) coverage;

[1]https://github.com/verimath/topology

iii) covering families. A category have one of these three is called a Grothendieck site.

Examples: Zariski, flat, étale, Nisnevich topologies.

A sheaf is a presheaf (functor from opposite category to category of sets) which satisties patching conditions arising from Grothendieck topology, and applying the associated sheaf functor to preashef forces compliance with these conditions.

The notion of Grothendieck topos is a geometric flavour of topos theory, where topos is defined as category of sheaves on a Grothendieck site with geometric moriphisms as adjoint pairs of functors between topoi, that satisfy exactness properties. [16]

As this flavour of topos theory uses category of sets as a prerequisite, the formal construction of set topos is cricual in doing sheaf topos theory.

Визначення 143. (Sieves). Sieves are a family of subfunctors

$$R \subset \mathrm{Hom}_C(,U), U \in C,$$

such that following axioms hold: i) (base change) If $R \subset \mathrm{Hom}_C(,U)$ is covering and $\phi : V \to U$ is a morphism of C, then the subfuntor

$$\phi^{-1}(R) = \{\gamma : W \to V \| \phi \cdot \gamma \in R\}$$

is covering for V; ii) (local character) Suppose that $R, R' \subset \mathrm{Hom}_C(,U)$ are subfunctors and R is covering. If $\phi^{-1}(R')$ is covering for all $\phi : V \to U$ in R, then R' is covering; iii) $\mathrm{Hom}_C(,U)$ is covering for all $U \in C$.

Визначення 144. (Coverage). A coverage is a function assigning to each Ob_C the family of morphisms $\{f_i : U_i \to U\}_{i \in I}$ called covering families, such that for any $g : V \to U$ exist a covering family $\{h : V_j \to V\}_{j \in J}$ such that each composite $h_j \circ g$ factors some f_i:

$$\begin{tikzcd} V_j \arrow[r, "k"] \arrow[d, "h"] & U_i \arrow[d, "f_i"] \\ V \arrow[r, "g"] & U \end{tikzcd}$$

```
Co (C: precategory) (cod: carrier C) : U
  = (dom: carrier C)
  * (hom C dom cod)

Delta (C: precategory) (d: carrier C) : U
  = (index: U)
  * (index -> Co C d)

Coverage (C: precategory): U
  = (cod: carrier C)
  * (fam: Delta C cod)
  * (coverings: carrier C -> Delta C cod -> U)
  * (coverings cod fam)
```

Визначення 145. (Grothendieck Topology). Suppose category C has all pullbacks. Since C is small, a pretopology on C consists of families of sets of morphisms

$$\{\phi_\alpha : U_\alpha \to U\}, U \in C,$$

called covering families, such that following axioms hold: i) suppose that $\phi_\alpha : U_\alpha \to U$ is a covering family and that $\psi : V \to U$ is a morphism of C. Then the collection $V \times_U U_\alpha \to V$ is a cvering family for V. ii) If $\{\phi_\alpha : U_\alpha \to U\}$ is covering, and $\{\gamma_{\alpha,\beta} : W_{\alpha,\beta} \to U_\alpha\}$ is covering for all α, then the family of composites

$$W_{\alpha,\beta} \xrightarrow{\gamma_{\alpha,\beta}} U_\alpha \xrightarrow{\phi_\alpha} U$$

is covering; iii) The family $\{1 : U \to U\}$ is covering for all $U \in C$.

Визначення 146. (Site). Site is a category having either a coverage, grothendieck topology, or sieves.

```
site (C: precategory): U
  = (C: precategory) * Coverage C
```

Визначення 147. (Presheaf). Presheaf of a category C is a functor from opposite category to category of sets: $C^{op} \to \mathrm{Set}$.

```
presheaf (C: precategory): U
  = catfunctor (opCat C) Set
```

Визначення 148. (Presheaf Category, **PSh**). Presheaf category **PSh** for a site C is category were objects are presheaves and morphisms are natural transformations of presheaf functors.

Визначення 149. (Sheaf). Sheaf is a presheaf on a site. In other words a presheaf $F : C^{op} \to \mathbf{Set}$ such that the cannonical map of inverse limit

$$F(U) \to \overleftarrow{\lim_{V \to U \in R}} F(V)$$

is an isomorphism for each covering sieve $R \subset Hom_C(\_, U)$. Equivalently, all induced functions

$$Hom_C(Hom_C(\_, U), F) \to Hom_C(R, F)$$

should be bejections.

```
sheaf (C: precategory): U
  = (S: site C)
  * presheaf S.1
```

Визначення 150. (Sheaf Category, **Sh**). Sheaf category **Sh** is a category where objects are sheaves and morphisms are natural transformation of sheves. Sheaf category is a full subcategory of category of presheaves **PSh**.

Визначення 151. (Grothendieck Topos). Topos is the category of sheaves $\mathbf{Sh}(C, J)$ on a site C with topology J.

Теорема 26. (Giraud). A category C is a Grothiendieck topos iff it has following properties: i) has all finite limits; ii) has small disjoint coproducts stable under pullbacks; iii) any epimorphism is coequalizer; iv) any equivalence relation $R \to E$ is a kernel pair and has a quotient; v) any coequalizer $R \to E \to Q$ is stably exact; vi) there is a set of objects that generates C.

Визначення 152. (Geometric Morphism). Suppose that C and D are Grothendieck sites. A geometric morphism

$$f : \mathbf{Sh}(C) \to \mathbf{Sh}(D)$$

consist of functors $f_* : \mathbf{Sh}(C) \to \mathbf{Sh}(D)$ and $f^* : \mathbf{Sh}(D) \to \mathbf{Sh}(C)$ such that f^* is left adjoint to f_* and f^* preserves finite limits. The left adjoint f^* is called the inverse image functor, while f_* is called the direct image. The inverse image functor f^* is left and right exact in the sense that it preserves all finite colimits and limits, respectively.

Визначення 153. (Cohesive Topos). A topos E is a cohesive topos over a base topos S, if there is a geometric morphism $(p^*, p_*) : E \to S$, such that: i) exists adjunction $p^! \vdash p_*$ and $p^! \dashv p_*$; ii) p^* and $p^!$ are full faithful; iii)

$p_!$ preserves finite products. This quadruple defines adjoint triple:

$$\int \dashv \flat \dashv \sharp$$

4.2.4. Елементарний топос. Giraud theorem was a synonymical topos definition involved only topos properties but not a site properties. That was step forward on predicative definition. The other step was made by Lawvere and Tierney, by removing explicit dependance on categorical model of set theory (as category of set is used in definition of presheaf). This information was hidden into subobject classifier which was well defined through categorical pullback and property of being cartesian closed (having lambda calculus as internal language).

Elementary topos doesn't involve 2-categorical modeling, so we can construct set topos without using functors and natural transformations (what we need in geometrical topos theory flavour). This flavour of topos theory more suited for logic needs rather that geometry, as its set properties are hidden under the predicative predicative pullback definition of subobject classifier rather that functorial notation of presheaf functor. So we can simplify proofs at the homotopy levels, not to lift everything to 2-categorical model.

Визначення 154. (Monomorphism). An morphism $f : Y \to Z$ is a monic or mono if for any object X and every pair of parralel morphisms $g_1, g_2 : X \to Y$ the

$$f \circ g_1 = f \circ g_2 \to g_1 = g_2.$$

More abstractly, f is mono if for any X the $\mathrm{Hom}(X, \_)$ takes it to an injective function between hom sets $\mathrm{Hom}(X, Y) \to \mathrm{Hom}(X, Z)$.

```
mono (P: precategory) (Y Z: carrier P) (f: hom P Y Z): U
   = (X: carrier P) (g1 g2: hom P X Y)
  -> Path (hom P X Z) (compose P X Y Z g1 f)
                      (compose P X Y Z g2 f)
  -> Path (hom P X Y) g1 g2
```

Визначення 155. (Subobject Classifier [17]). In category C with finite limits, a subobject classifier is a monomorphism $\mathsf{true} : 1 \to \Omega$ out of terminal object 1, such that for any mono $U \to X$ there is a unique morphism $\chi_U : X \to \Omega$ and pullback diagram:

$$\begin{tikzcd} U \arrow[r, "k"] \arrow[d] & 1 \arrow[d, "\mathsf{true}"] \\ X\Omega \arrow[r, "\chi_U"] & \Omega \end{tikzcd}$$

```
subobjectClassifier (C: precategory): U
  = (omega: carrier C)
  * (end: terminal C)
  * (trueHom: hom C end.1 omega)
  * (chi: (V X: carrier C) (j: hom C V X) -> hom C X omega)
  * (square: (V X: carrier C) (j: hom C V X) -> mono C V X j
    -> hasPullback C (omega,(end.1,trueHom),(X,chi V X j)))
  * ((V X: carrier C) (j: hom C V X) (k: hom C X omega)
    -> mono C V X j
    -> hasPullback C (omega,(end.1,trueHom),(X,k))
    -> Path (hom C X omega) (chi V X j) k)
```

Теорема 27. (Category of Sets has Subobject Classifier).

Визначення 156. (Cartesian Closed Categories). The category C is called cartesian closed if exists all: i) terminals; ii) products; iii) exponentials. Note that this definition lacks beta and eta rules which could be found in embedding MLTT.

```
isCCC (C: precategory): U
  = (Exp:   (A B: carrier C) -> carrier C)
  * (Prod:  (A B: carrier C) -> carrier C)
  * (Apply: (A B: carrier C) -> hom C (Prod (Exp A B) A) B)
  * (P1:    (A B: carrier C) -> hom C (Prod A B) A)
  * (P2:    (A B: carrier C) -> hom C (Prod A B) B)
  * (Term:  terminal C)
  * unit
```

Теорема 28. (Category of Sets is cartesian closed). As you can see from exp and pro we internalize Π and Σ types as SET instances, the isSet predicates are provided with contractability. Exitense of terminals is proved by propPi. The same technique you can find in **MLTT** embedding.

```
cartesianClosure : isCCC Set
  = (expo,prod,appli,proj1,proj2,term,tt) where
    exp (A B: SET): SET = (A.1   -> B.1, setFun A.1 B.1 B.2)
    pro (A B: SET): SET = (prod A.1 B.1, setSig A.1 (\(_ : A.1)
                          -> B.1) A.2 (\(_ : A.1) -> B.2))
    expo:  (A B: SET) -> SET = \(A B: SET) -> exp A B
    prod:  (A B: SET) -> SET = \(A B: SET) -> pro A B
    appli: (A B: SET) -> hom Set (pro (exp A B) A) B
        = \(A B: SET) -> \(x:(pro(exp A B)A).1)-> x.1 x.2
    proj1: (A B: SET) -> hom Set (pro A B) A
        = \(A B: SET) (x: (pro A B).1) -> x.1
    proj2: (A B: SET) -> hom Set (pro A B) B
        = \(A B: SET) (x: (pro A B).1) -> x.2
    unitContr (x: SET) (f: x.1 -> unit) : isContr (x.1 -> unit)
      = (f, \(z: x.1 -> unit) -> propPi x.1 (\(_:x.1)->unit)
             (\(x:x.1) -> propUnit) f z)
    term: terminal Set = ((unit,setUnit),
             \(x: SET) -> unitContr x (\(z: x.1) -> tt))
```

Note that rules of cartesian closure forms a type theoretical langage called lambda calculus.

Визначення 157. (Elementary Topos). Topos is a precategory which is cartesian closed and has subobject classifier.

```
Topos (cat: precategory) : U
  = (cartesianClosure: isCCC cat)
  * subobjectClassifier cat
```

Теорема 29. (Topos Definitions). Any Grothendieck topos is an elementary topos too. The proof is sligthly based on results of Giraud theorem.

Теорема 30. (Category of Sets forms a Topos). There is a cartesian closure and subobject classifier for a categoty of sets.

```
internal : Topos Set
         = (cartesianClosure,hasSubobject)
```

Теорема 31. (Freyd). Main theorem of topos theory [18]. For any topos C and any $b : Ob_C$ relative category $C \downarrow b$ is also a topos. And for any arrow $f : a \to b$ inverse image functor $f^* : C \downarrow b \to c \downarrow a$ has left adjoint $\sum_f$ and right adjoin $\prod_f$.

4.2.5. Висновки. We gave here constructive definition of topology as finite unions and intersections of open subsets. Then make this definition categorically compatible by introducing Grothendieck topology in three different forms: sieves, coverage, and covering families. Then we defined an elementary topos and introduce category of sets, and proved that **Set** is cartesian closed, has object classifier and thus a topos.

This intro could be considered as a formal introduction to topos theory (at least of the level of first chapter) and you may evolve this library to your needs or ask to help porting or developing your application of topos theory to a particular formal construction.

4.3. Алгебраїчна топологія

4.3.1. Теорія груп.

4.3.2. Простори.

4.3.2.1. Сімпліціальні.

4.3.2.2. CW-комплекси. The definition of homotopy groups, a special role is played by the inclusions $S^{n-1} \hookrightarrow D^n$. We study spaces obtained iterated attachments of D^n along S^{n-1}.

Визначення 158. (Attachment). Attaching n-cell to a space X along a map $f : S^{n-1} \to X$ means taking a pushout figure.

$$\begin{tikzcd} S^{n-1} \arrow[r, "k"] \arrow[d] & X \arrow[d] \\ D^n \arrow[r, "g"] & \cup_f D^n \end{tikzcd}$$

where the notation $X \cup_f D^n$ means result depends on homotopy class of f.

Визначення 159. (CW-Complex). Inductively. The only CW-complex of dimention -1 is $\varnothing$. A CW-complex of dimension $\leqslant n$ on X is a space X obtained by attaching a collection of n-cells to a CW-complex of dimension $n-1$.

A CW-complex is a space X which is the $\mathtt{colimit}(X_i)$ of a sequence $X_{-1} = \varnothing \hookrightarrow X_0 \hookrightarrow X_1 \hookrightarrow X_2 \hookrightarrow ...X$ of CW-complexes X_i of dimension $\leqslant n$, with X_{i+1} obtained from X_i by i-cell attachments. Thus if X is a CW-complex, it comes with a filtration

$$\varnothing \hookrightarrow X_0 \hookrightarrow X_1 \hookrightarrow X_2 \hookrightarrow ...X$$

where X_i is a CW-complex of dimension $\leqslant i$ called the i-skeleton, and hence the filtration is called the skeletal filtration.

4.3.3. Теорія гомотопій.

4.3.3.1. Простори петель.

Визначення 160. (Pointed Space). A pointed type (A, a) is a type A : U together with a point $a : A$, called its basepoint.

```
pointed: U = (A: U) * A
point (A: pointed): A.1 = A.2
space (A: pointed): U = A.1
```

Визначення 161. (Loop Space).

$$\Omega(A, a) =_{\mathrm{def}} ((a =_A a), \mathtt{refl}_A(a)).$$

```
omega1 (A: pointed) : pointed
  = (Path (space A) (point A) (point A), refl A.1 (point A))
```

Визначення 162. (n-Loop Space).

$$\begin{cases} \Omega^0(A, a) =_{\mathrm{def}} (A, a) \\ \Omega^{n+1}(A, a) =_{\mathrm{def}} \Omega^n(\Omega(A, a)) \end{cases}$$

```
omega : nat -> pointed -> pointed = split
  zero -> idfun pointed
  succ n -> \(A: pointed) -> omega n (omega1 A)
```

4.3.3.2. Обчислення гомотопічних груп.

Визначення 163. (n-th Homotopy Group of m-Sphere).

$$\pi_n S^m = \|\Omega^n(S^m)\|_0.$$

```
piS (n: nat): (m: nat) -> U = split
   zero   -> sTrunc (space (omega n (bool,false)))
   succ x -> sTrunc (space (omega n (Sn (succ x),north)))
```

Теорема 32. ($\Omega(S^1) = \mathbb{Z}$).

```
data S1 = base
        | loop <i> [ (i=0) -> base ,
                     (i=1) -> base ]

loopS1 : U = Path S1 base base

encode (x:S1) (p:Path S1 base x)
  : helix x
  = subst S1 helix base x p zeroZ

decode : (x:S1) -> helix x -> Path S1 base x = split
  base -> loopIt
  loop @ i -> rem @ i where
    p : Path U (Z -> loopS1) (Z -> loopS1)
      = <j> helix (loop1@j) -> Path S1 base (loop1@j)
    rem : PathP p loopIt loopIt
      = corFib1 S1 helix (\(x:S1)->Path S1 base x) base
        loopIt loopIt loop1 (\(n:Z) ->
        comp (<i> Path loopS1 (oneTurn (loopIt n))
```

```
            (loopIt (testIsoPath Z Z sucZ predZ
                     sucpredZ predsucZ n @ i)))
            (<i>(lem1It n)@-i) [])

loopS1eqZ : Path U Z loopS1
  = isoPath Z loopS1 (decode base) (encode base)
    sectionZ retractZ
```

4.3.3.3. Розшарування Хопфа. This article defines the Hopf Fibration (HF), the concept of splitting the S^3 sphere onto the twisted cartesian product of spheres S^1 and S^2. Basic HF applications are: 1) HF is a Fiber Bundle structure of Dirac Monopole; 2) HF is a map from the S^3 in H to the Bloch sphere; 3) If HF is a vector field in R^3 then there exists a solution to compressible non-viscous Navier-Stokes equations. It was figured out that there are only 4 dimensions of fibers with Hopf invariant 1, namely S^0, S^1, S^3, S^7.

This article consists of two parts: 1) geometric visualization of projection of S^3 to S^2 (frontend); 2) formal topological version of HF in cubical type theory (backend). Consider this a basic intro and a summary of results or companion guide to the chapter 8.5 from HoTT.

Геометрична інтерпретація. Let's imagine S^3 as smooth differentiable manifold and build a projection onto the display as if demoscene is still alive.

Визначення 164. (Sphere S^3). Like a little baby in $\mathbb{R}^4$:

$$S^3 = \{(x_0, x_1, x_2, x_3) \in \mathbb{R}^4 : \sum_{i=0}^{3} x_i^2 = 1\};$$

after math classes in quaternions $\mathbb{H}$:

$$S^3 = \{x \in \mathbb{H} : \|x\| = 1\}.$$

Визначення 165. (Locus). The S^3 is realized as a disjoint union of circular fibers in Hopf coordinates $(\eta, \theta_1, \theta_2)$:

$$\begin{cases} x_0 = \cos(\theta_1)\sin(\eta), \\ x_1 = \sin(\theta_1)\sin(\eta), \\ x_2 = \cos(\theta_2)\cos(\eta), \\ x_3 = \sin(\theta_2)\cos(\eta). \end{cases}$$

Where $\eta \in [0, \frac{\pi}{2}]$ and $\theta_{1,2} \in [0, 2\pi]$.

Визначення 166. (Mapping on S^2). A mapping of the Locus to the S^2 has points on the circles parametrized by θ_2:

$$\begin{cases} x = \sin(2\eta)\cos(\theta_1), \\ y = \sin(2\eta)\sin(\theta_1), \\ z = \cos(2\eta). \end{cases}$$

```
var fiber = new THREE.Curve(),
    color = sphericalCoords.color;

fiber.getPoint = function(t) {
    var eta = sphericalCoords.eta,
        phi = sphericalCoords.phi,
        theta = 2 * Math.PI * t;
    var x1 = Math.cos(phi+theta) * Math.sin(eta/2),
```

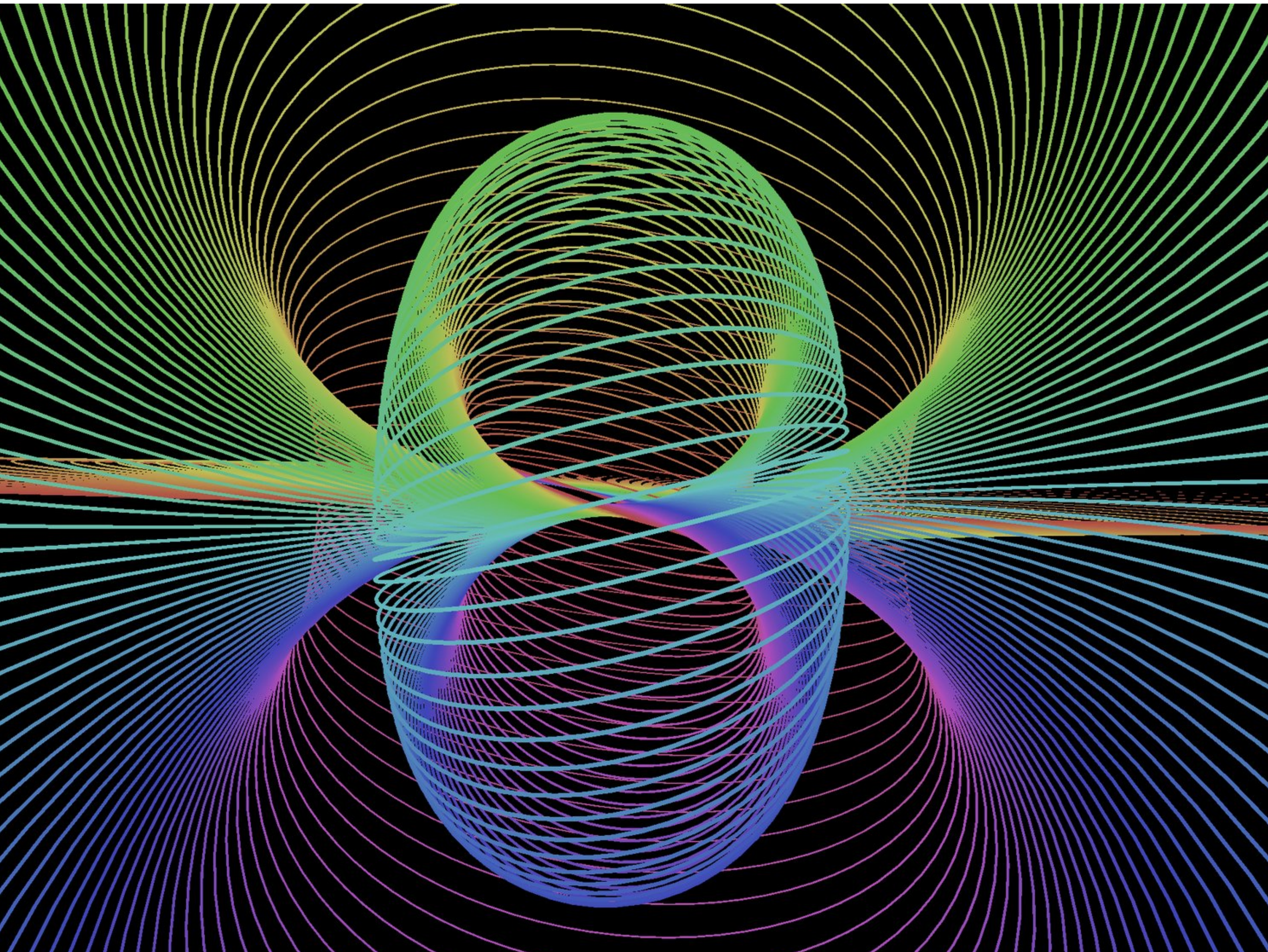

Рис. 4.1. Розшарування Хопфа

```
        x2 = Math.sin(phi+theta) * Math.sin(eta/2),
        x3 = Math.cos(phi-theta) * Math.cos(eta/2),
        x4 = Math.sin(phi-theta) * Math.cos(eta/2);
    var m = mag([x1,x2,x3]),
        r = Math.sqrt((1-x4)/(1+x4));
        return new THREE.Vector3(r*x1/m,r*x2/m, r*x3/m);
    };
```

Гомотопічна інтерпретація. Can we reason about spheres without a metric? Yes! But can we do this in a constructive way? Also yes.

4.3.3.4. Розшарування Хопфа.

Приклад 3. (S^3 Hopf Fiber). Вперше розшарування Хопфа трьовимірної сфери було формалізовано Джильямом Брунері. Тут дається його модифікована версія.

```
rot: (x : S1) -> Path S1 x x = split
    base -> loop1
    loop @ i -> constSquare S1 base loop1 @ i

mu : S1 -> equiv S1 S1 = split
    base -> idEquiv S1
    loop @ i -> equivPath S1 S1 (idEquiv S1)
                (idEquiv S1) (<j> \(x : S1) -> rot x @ j) @ i

H : S2 -> U = split
    north -> S1
    south -> S1
    merid x @ i -> ua S1 S1 (mu x) @ i

total : U = (c : S2) * H c
```

Визначення 167. (H-space). H-space over a carrier A is a tuple

$$H_A = \begin{cases} A : U \\ e : A \\ \mu : A \to A \to A \\ \beta : (a : A) \to \Sigma(\mu(e, a) = a)(\mu(a, e) = a) \end{cases}$$

.

Теорема 33. (Hopf Fibrations). There are fiber bundles: (S^0, S^1, p, S^1), (S^1, S^3, p, S^2), (S^3, S^7, p, S^4), (S^7, S^{15}, p, S^8).

Визначення 168. (Hopf Invariant). Let $\phi : S^{2n-1} \to S^n$ a continuous map. Then homotopy pushout (cofiber) of ϕ is $\mathtt{cofib}(\phi) = S^n \bigcup_\phi \mathbb{D}^{2n}$ has ordinary cohomology

$$H^k(\mathtt{cofib}(\phi), \mathbb{Z}) = \begin{cases} \mathbb{Z} \text{ for } k = n, 2n \\ 0 \text{ otherwise} \end{cases}$$

Hence for α, β generators of the cohomology groups in degree n and $2n$, respectively, there exists an integer $h(\phi)$ that expresses the **cup product** square of α as a multiple of β — $\alpha \sqcup \alpha = h(\phi) \cdot \beta$. This integer $h(\phi)$ is called Hopf invariant of ϕ.

Теорема 34. (Adams, Atiyah). Hopf Fibrations are only maps that have Hopf invariant 1.

4.3.4. Теорія гомологій.

4.4. Диференціальна геометрія

4.4.1. V-многовиди.

4.4.2. G-структури.

4.4.3. H-простори.

РОЗДІЛ 5
ДОДАТКИ

У додатках ми використаємо три різних мови, та покаже два застосування формальних мов: 1) формальна філософія (на мові O_{HTS}; 2) формальний ввід-вивід для системної інженерії (на двох промислових мовах Erlang та OCaml).

5.1. Формалізація вводу-виводу для OCaml

```
CoInductive Co (E : Effect.t) : Type -> Type :=
    | Bind : forall (A B : Type), Co E A -> (A -> Co E B) -> Co E B
    | Split : forall (A : Type), Co E A -> Co E A -> Co E A
    | Join : forall (A B : Type), Co E A -> Co E B -> Co E (A * B).
    | Ret : forall (A : Type) (x : A), Co E A
    | Call : forall (command : Effect.command E),
                 Co E (Effect.answer E command)

Definition run (argv : list LString.t): Co effect unit :=
    ido! log (LString.s "What is your name?") in
    ilet! name := read_line in
    match name with
      | None => ret tt
      | Some name => log (LString.s "Hello " ++ name ++ LString.s "!")
    end.

Parameter infinity : nat.
Definition eval {A} (x : Co effect A) : Lwt.t A := eval_aux infinity x.
```

```
Fixpoint eval_aux {A} (steps : nat) (x : Co effect A) : Lwt.t A :=
    match steps with
      | O => error tt
      | S steps =>
        match x with
          | Bind _ _ x f => Lwt.bind   (eval_aux steps x)
                              (fun v_x => eval_aux steps (f v_x))
          | Split _ x y  => Lwt.choose (eval_aux steps x)
                                       (eval_aux steps y)
          | Join _ _ x y => Lwt.join   (eval_aux steps x)
                                       (eval_aux steps y)
          | Ret _ v => Lwt.ret v
          | Call c => eval_command c
        end
    end.

CoFixpoint handle_commands : Co effect unit :=
    ilet! name := read_line in
    match name with
      | None => ret tt
      | Some command =>
        ilet! result := log (LString.s "Input: "
                     ++ command ++ LString.s ".")
     in handle_commands
    end.

Definition launch (m: list LString.t -> Co effect unit): unit :=
    let argv := List.map String.to_lstring Sys.argv in
    Lwt.launch (eval (m argv)).

Definition corun (argv: list LString.t): Co effect unit :=
    handle_commands.

Definition main := launch corun.
```

5.2. Формалізація вводу-виводу для Erlang

This work is expected to compile to a limited number of target platforms. For now, Erlang, Haskell, and LLVM are awaiting. Erlang version is expected to be used both on LING and BEAM Erlang virtual machines. This language allows you to define trusted operations in System F and extract this routine to Erlang/OTP platform and plug as trusted resources. As the example, we also provide infinite coinductive process creation and inductive shell that linked to Erlang/OTP IO functions directly.

IO protocol. We can construct in pure type system the state machine based on (co)free monads driven by **IO/IOI** protocols. Assume that **String** is a **List Nat** (as it is in Erlang natively), and three external constructors: getLine, putLine and pure. We need to put correspondent implementations on host platform as parameters to perform the actual IO.

```
String: Type = List Nat
data IO: Type =
     (getLine: (String -> IO) -> IO)
     (putLine: String -> IO)
     (pure: () -> IO)
```

5.2.0.1. Infinity I/O Type. Infinity I/O Type Spec.

```
-- IOI/@: (r: U) [x: U] [[s: U] -> s -> [s -> #IOI/F r s] -> x] x
   \ (r : *)
-> \/ (x : *)
-> (\/ (s : *)
   -> s
   -> (s -> #IOI/F r s)
   -> x)
-> x

-- IOI/F
   \ (a : *)
-> \ (State : *)
-> \/ (IOF : *)
-> \/ (PutLine_ : #IOI/data -> State -> IOF)
-> \/ (GetLine_ : (#IOI/data -> State) -> IOF)
-> \/ (Pure_ : a -> IOF)
-> IOF

-- IOI/MkIO
   \ (r : *)
-> \ (s : *)
-> \ (seed : s)
-> \ (step : s -> #IOI/F r s)
-> \ (x : *)
-> \ (k : forall (s : *) -> s -> (s -> #IOI/F r s) -> x)
-> k s seed step

-- IOI/data
#List/@ #Nat/@
```

Infinite I/O Sample Program.

```
-- Morte/corecursive
( \ (r: *1)
 -> ( (((#IOI/MkIO r) (#Maybe/@ #IOI/data)) (#Maybe/Nothing #IOI/data))
    ( \ (m: (#Maybe/@ #IOI/data))
     -> (((((#Maybe/maybe #IOI/data) m) ((#IOI/F r) (#Maybe/@ #IOI/data)))
           ( \ (str: #IOI/data)
            -> ((((#IOI/putLine r) (#Maybe/@ #IOI/data)) str)
                 (#Maybe/Nothing #IOI/data))))
         (((#IOI/getLine r) (#Maybe/@ #IOI/data))
          (#Maybe/Just #IOI/data))))))
```

Erlang Coinductive Bindings.

```
copure() ->
    fun (_) -> fun (IO) -> IO end end.

cogetLine() ->
    fun(IO) -> fun(_) ->
        L = ch:list(io:get_line("> ")),
        ch:ap(IO,[L]) end end.

coputLine() ->
    fun (S) -> fun(IO) ->
        X = ch:unlist(S),
        io:put_chars(": "++X),
        case X of "0\n" -> list([]);
                      _ -> corec() end end end.

corec() ->
    ap('Morte':corecursive(),
        [copure(),cogetLine(),coputLine(),copure(),list([])]).
```

```
> om_extract:extract("priv/normal/IOI").
ok
> Active: module loaded: {reloaded,'IOI'}

> om:corec().
> 1
: 1
> 0
: 0
#Fun<List.3.113171260>
```

5.2.0.2. I/O Type.

I/O Type Spec.

```
-- IO/@
   \ (a : *)
-> \/ (IO : *)
-> \/ (GetLine_ : (#IO/data -> IO) -> IO)
-> \/ (PutLine_ : #IO/data -> IO -> IO)
-> \/ (Pure_ : a -> IO)
-> IO

-- IO/replicateM
   \ (n: #Nat/@)
-> \ (io: #IO/@ #Unit/@)
-> #Nat/fold n (#IO/@ #Unit/@)
               (#IO/[>>] io)
               (#IO/pure #Unit/@ #Unit/Make)
```

Guarded Recursion I/O Sample Program.

```
-- Morte/recursive
((#IO/replicateM #Nat/Five)
 ((((#IO/[>>=] #IO/data) #Unit/@) #IO/getLine) #IO/putLine))
```

Erlang Inductive Bindings.

```
pure() ->
    fun(IO) -> IO end.

getLine() ->
    fun(IO) -> fun(_) ->
        L = ch:list(io:get_line("> ")),
        ch:ap(IO,[L]) end end.

putLine() ->
    fun (S) -> fun(IO) ->
        io:put_chars(": "++ch:unlist(S)),
        ch:ap(IO,[S]) end end.

rec() ->
    ap('Morte':recursive(),
        [getLine(),putLine(),pure(),list([])]).
```

Here is example of Erlang/OTP shell running recursive example.

```
> om:rec().
> 1
: 1
> 2
: 2
> 3
: 3
> 4
: 4
> 5
: 5
#Fun<List.28.113171260>
```

5.3. Формалізація мадх'яміки

Сейчас я дам вам почувствовать вкус формальной философии по-настоящему! А то, вам может показаться, что это канал по формальной математике, а не формальной философии. Я же считаю, что если формальная философия не опирается на формальную математику, то грош цена такой формальной философии.

```
module buddhism where
import path
```

Сегодня мы будем формализировать понятие недвойственности в буддизме, которое связано сразу со многими концепциями на уровнях Сутры, Тантры и Дзогчена: понятием взаимозависимого возникновения и понятием пустотности всех феноменов (Сутра Праджняпарамиты). Классический пример с расчленением тела ставит вопрос, когда тело перестает быть человеком-существом, если от него начать отрубывать куски мяса (мы буддисты любим и лилеем такие мысленные образы-эксперименты) или другими словами, чтобы отличить тело от не-тела, нам нужен двуместный предикат (семья типов), функция, которая может идентифицировать конректные два эклемпляры тела. Фактически речь идет об индентификации двух объектов, т.е. обычном типе-равенстве Мартина-Лёфа.

За фреймворк возьмем концепты Готтлоба Фреге, согласно определению, концепт – это предикат над объектом или, другими словами, Пи-тип Мартина-Лёфа, индексированный тип, семья типов, тривиальное расслоение и т.д. Где объект x из o принадлежит концепту, только если сам концепт, параметризированный этим объектом, населен p(o) : U (где p : concept o).

```
concept (o: U): U
  = o -> U
```

Концепт p должен предоставлять пример или контрпример в различении, т.е. чтобы определить тело это или не-тело еще, пока мы его расчленяем, нам нужно как минимум два куска: тело и не-тело как примеры идентификации. Таким образом, недвойственность может быть представлена, как равенство между всеми расслоениями (предекатами над объектами).

```
nondual (o: U) (p: concept o): U
  = (x y: o) -> Path U (p x) (p y)
```

Итак, недвойственность устраняет различие между примерами и контрпримерами на примордиальном уровне мандалы MLTT, иными словами идентифицирует все концепты. Сама же идентификация классов объектов, которые принадлежат разным концептам — это условие, сжимающее все объекты в точку, или стягиваемое пространство, вершина конуса мандалы MLTT, или, другими словами, пустотность всех феноменов.

```
allpaths (o: U): U
  = (x y: o) -> Path o x y
```

Формулировка буддийской теоремы недвойственности которая распространяется для всех типов учеников (тупых, средних и смышленых), может звучать так: недвойственность концепта есть способ идентификации его объектов. Сформулируем эту же теорему в другую сторону: способ идентификации объектов задает предикат недвойственности концептов. Туда — ((p: concept o) -> nondual o p) -> allpaths o, Сюда — allpaths o -> ((p: concept o) -> nondual o p). И докажем ее! Как видно из сигнатур нам всего-лишь надо построить функцию транспорта между двумя пространствами путей: (p x) $=_U$ (p y) и x $=_o$ y. Воспользуемся приведением пути в стрелку (coerce) и конгруэнтностью (cong) из базовой библиотеки.

```
encode (o:U): ((p: concept o) -> nondual o p) -> allpaths o
  = \(nd: (p: concept o) -> nondual o p) (a b: o)
  -> coerce(Path o a a)(Path o a b)(nd(\(z:o)->Path o a z)a b)(refl o a)

decode (o:U): allpaths o -> ((p: concept o) -> nondual o p)
  = \(all: allpaths o)(p: concept o)(x y: o) -> cong o U p x y (all x y)
```

Как видите, теоремка о пустотности всех феноменов получилась на пару строчек, которые демонстрируют: 1) основы формальной философии и быстрый вкат; 2) хороший пример к первой главе HoTT на пространство путей и модуль path.

[19] [2] [3] [20] [5] [6] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [4] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57]

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] N. G. de Bruijn, *AUTOMATH, a Language for Mathematics*, pp. 159–200. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983.

[2] P. Martin-Löf and G. Sambin, *Intuitionistic type theory*. Studies in proof theory, Bibliopolis, 1984.

[3] T. Coquand and G. Huet, "The calculus of constructions," in *Information and Computation*, (Duluth, MN, USA), pp. 95–120, Academic Press, Inc., 1988.

[4] F. Pfenning and C. Paulin-Mohring, "Inductively defined types in the calculus of constructions," in *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*, pp. 209–228, 1989.

[5] H. P. Barendregt, "Lambda calculi with types," in *Handbook of Logic in Computer Science (Vol. 2)* (S. Abramsky, D. M. Gabbay, and S. E. Maibaum, eds.), (New York, NY, USA), pp. 117–309, Oxford University Press, Inc., 1992.

[6] S. P. Jones and E. Meijer, "Henk: A typed intermediate language," in *In Proc. First Int'l Workshop on Types in Compilation*, 1997.

[7] C.-E. Ore, "The extended calculus of constructions (ecc) with inductive types," in *Information and Computation*, vol. 99, pp. 231

– 264, 1992.

[8] M. Sokhatskyi and P. Maslianko, "The systems engineering of consistent pure language with effect type system for certified applications and higher languages," in *AIP Conference Proceedings*, vol. 1982, p. 020033, AIP Publishing, 2018.

[9] P. Fu and A. Stump, "Self types for dependently typed lambda encodings," in *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pp. 224–239, 2014.

[10] A. Stump, "The calculus of dependent lambda eliminations," in *Journal of Functional Programming*, vol. 27, Cambridge University Press, 2017.

[11] G. Barthe, *Extensions of pure type systems*, pp. 16–31. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995.

[12] P. Martin-Löf, "An intuitionistic theory of types: Predicative part," in *Studies in Logic and the Foundations of Mathematics*, vol. 80, pp. 73–118, Elsevier, 1975.

[13] H. Geuvers, *Induction Is Not Derivable in Second Order Dependent Type Theory*, pp. 166–181. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[14] S. Awodey, "Impredicative encodings in hott," 2017.

[15] S. Speight, "Impredicative encoding of inductive types in hott," 2017.

[16] J. Jardine, *Local Homotopy Theory*. Springer Monographs in

Mathematics, Springer New York, 2015.

[17] P. Johnstone, *Topos Theory*. Dover Books on Mathematics, Dover Publications, 2014.

[18] R. Goldblatt, *Topoi: The Categorial Analysis of Logic*. Studies in Logic and the Foundations of Mathematics, Elsevier Science, 2014.

[19] P. Martin-Löf and G. Sambin, *The Theory of Types*. Studies in proof theory, 1972.

[20] M. Hofmann and T. Streicher, "The groupoid interpretation of type theory," in *In Venice Festschrift*, pp. 83–111, Oxford University Press, 1996.

[21] C. Hermida and B. Jacobs, "Fibrations with indeterminates: Contextual and functional completeness for polymorphic lambda calculi," *Mathematical Structures in Computer Science*, vol. 5, pp. 501–531, 1995.

[22] P.-L. Curien, "Category theory: a programming language-oriented introduction," 2008.

[23] S. MacLane, *Categories for the Working Mathematician*. New York: Springer-Verlag, 1971. Graduate Texts in Mathematics, Vol. 5.

[24] F. Lawvere and S. Schanuel, *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, 2009.

[25] A. Buisse and P. Dybjer, "The interpretation of intuitionistic type theory in locally cartesian closed categories – an intuitionistic perspective," *Electron. Notes Theor. Comput. Sci.*, vol. 218, pp. 21–32, Oct. 2008.

[26] P. Clairambault, "From categories with families to locally cartesian closed categories,"

[27] A. Abel, T. Coquand, and P. Dybjer, "On the algebraic foundation of proof assistants for intuitionistic type theory," in *Functional and Logic Programming* (J. Garrigue and M. V. Hermenegildo, eds.), (Berlin, Heidelberg), pp. 3–13, Springer Berlin Heidelberg, 2008.

[28] R. A. Seely, "Locally cartesian closed categories and type theory," in *Mathematical proceedings of the Cambridge philosophical society*, vol. 95, pp. 33–48, Cambridge University Press, 1984.

[29] P.-L. Curien, R. Garner, and M. Hofmann, "Revisiting the categorical interpretation of dependent type theory," *Theoretical Computer Science*, vol. 546, pp. 99–119, 2014.

[30] S. Castellan, "Dependent type theory as the initial category with families," *Internship Report*, 2014.

[31] V. Voevodsky, "A c-system defined by a universe in a category," 2014.

[32] P. Dybjer, "Internal type theory," in *International Workshop on Types for Proofs and Programs*, pp. 120–134, Springer, 1995.

[33] E. Bishop, *Foundations of constructive analysis*. 1967.

[34] B. Nordström, K. Petersson, and J. M. Smith, *Programming in Martin-Löf's type theory*, vol. 200. Oxford University Press Oxford, 1990.

[35] C. Hermida and B. Jacobs, "Structural induction and coinduction in a fibrational setting," *Information and computation*, vol. 145, no. 2, pp. 107–152, 1998.

[36] G. Barthe, V. Capretta, and O. Pons, "Setoids in type theory," 2000.

[37] V. Voevodsky, “A c-system defined by a universe category,” *Theory Appl. Categ*, vol. 30, no. 37, pp. 1181–1215, 2015.

[38] M. Sozeau and N. Tabareau, “Internalizing intensional type theory,”

[39] D. Selsam and L. de Moura, “Congruence closure in intensional type theory,” in *International Joint Conference on Automated Reasoning*, pp. 99–115, Springer, 2016.

[40] C. Böhm and A. Berarducci, “Automatic synthesis of typed lambda-programs on term algebras,” in *Theoretical Computer Science*, vol. 39, pp. 135–154, 1985.

[41] P. Wadler in *Recursive types for free*, manuscript, 1990.

[42] N. Gambino and M. Hyland, “Wellfounded trees and dependent polynomial functors,” in *International Workshop on Types for Proofs and Programs*, pp. 210–225, Springer, 2003.

[43] P. Dybjer in *Inductive families*, vol. 6, pp. 440–465, Springer, 1994.

[44] B. Jacobs and J. Rutten in *A tutorial on (co) algebras and (co) induction*, vol. 62, pp. 222–259, EUROPEAN ASSOCIATION FOR THEORETICAL COMPUTER, 1997.

[45] V. Vene, *Categorical programming with inductive and coinductive types*. Tartu University Press, 2000.

[46] H. Basold and H. Geuvers, “Type theory based on dependent inductive and coinductive types,” in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pp. 327–336, ACM, 2016.

[47] M. Hofmann and T. Streicher, “The groupoid model refutes uniqueness of identity proofs,” in *Logic in Computer Science, 1994.*

LICS'94. Proceedings., Symposium on, pp. 208–212, IEEE, 1994.

[48] B. Jacobs, *Categorical logic and type theory*, vol. 141. Elsevier, 1999.

[49] A. Joyal, "Categorical homotopy type theory," *Slides from a talk at MIT dated*, vol. 17, 2014.

[50] T. Coquand, P. Martin-Löf, V. Voevodsky, A. Joyal, A. Bauer, S. Awodey, M. Sozeau, M. Shulman, D. Licata, Y. Bertot, P. Dybjer, and N. Gambino, *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013.

[51] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg in *Cubical Type Theory: a constructive interpretation of the univalence axiom*, vol. abs/1611.02108, 2017.

[52] B. Ahrens, K. Kapulkin, and M. Shulman, "Univalent categories and the rezk completion," in *Extended Abstracts Fall 2013* (M. d. M. González, P. C. Yang, N. Gambino, and J. Kock, eds.), (Cham), pp. 75–76, Springer International Publishing, 2015.

[53] I. Orton and A. M. Pitts, "Axioms for modelling cubical type theory in a topos," *arXiv preprint arXiv:1712.04864*, 2017.

[54] S. Huber, "Cubical intepretations of type theory," 2016.

[55] S. Huber, "Canonicity for cubical type theory," *Journal of Automated Reasoning*, pp. 1–38, 2017.

[56] C. Angiuli, R. Harper, and T. Wilson, "Computational higher type theory i: Abstract cubical realizability," *arXiv preprint arXiv:1604.08873*, 2016.

[57] C. Angiuli and R. Harper, "Computational higher type theory ii:

Dependent cubical realizability," *arXiv preprint arXiv:1606.09638*, 2016.