Volume V: Design

Verification System Design and Implementation

Namdak Tonpa

2020 · Groupoid Infinity

V

Зміст

ВСТУП

Присвячується піонерам
формальної школи філософії

Фреге, Расселу, Вайтхеду,
Геделю, Гільберту, Каррі,
Чорчу

У вступі розказується про новий формальний підхід до математичної верифікації і спробу автора у цій парадигмі побудувати замкнену уніфіковану систему формальних мов для програмування, математики і філософії. В процесі розробки моделі такої системи автору довелося апробувати частини її імплементації для головних SML-подібних формальних академічних мов, мови Erlang і інших (загалом 7 мов). За 10 років автором було проаналізовані синтаксис і семантика основних мов програмування (більше 50 мов) з різних промислових і академічних доменів, 8 мов з яких були особисто реалізовані автором. В роботі описані 8 мов уніфікованої мовної системи (концептуальна модель) і представлені 2 їх імплементації.

Головним чином, натхнення було почерпнуте з LISP-машин минулого, APL-систем, перших систем доведення теорем таких як AUTOMATH, віртуальних маших паралельної і узгодженої обробки нескінченних процесів, таких як BEAM, кубічних MLTT-пруверів.

Вступне слово

Якщо говорити про математичну логіку, формальну математику, формальні методи, то основоположниками цих теорій можно вважати Бертрана Рассела і Альфреда Норта Вайтхеда, зокремаїх роботу Principia Mathematica, де будується формально теорія множин і доводиться твердження 1+1=2. Пізніше методи і предмет лябда-числення були розроблені Хаскелем Каррі і Алонсо Черчем, а теорема Геделя про неповноту до цих пір знаходить своє відображення в інфініті-топосах та у зліченних всесвітах сучасних пруверів.

Зараз формальні методи верифікації та відповідно теорії на яких вони побудовані є актуальними засобами забезпечення математичної якості

та гарантій для розробки не тільки програмного забезпечення, але і математики, і, навіть, формальної філософії.

0.1 Актуальність роботи

Обґрунтування вибору теми дослідження зобумовлене високою ціною помилок в складних системах.

Основні відомі приклади високої ціни помилок в індустрії програмного забезпечення: 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) 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.

Таким чином зросла популярність формальних мов з типизацією, які дають певні гарантії на стадії компіляції, витрачаючи при цьому небагато часу на специфікації. Піонер у цьому класі мов — Standard ML та ML-подібні мови.

Пізніше, більш загально, Системи Жирара F та F_ω стали промисловим стандартом де-факто. Усі сучасні мови для популярних віртуальних машин та середовищ виконання намагаються бути близькими до цих типових систем. Для JVM існує мова Scala, але перша формальна мова для JVM був порт Standard ML — MLj[7]. Для CLR існує мова F#. Мова Haskell поставляється разом з середовищем виконання GHC. Усі ці мови так чи інакше пов'язані з системами Жирара.

Ціна помилок в математичних доведеннях теж актуальна та дотична до теми цієї роботи — формальної верифікації. Особливо великі доведення які складаються з багатьох сотень сторінок вимагають ретельної та кропіткої

[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`

[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`

[7] `https://www.dcs.ed.ac.uk/home/mlj/`

роботи рецензентів, що може бути замінено формальними доведеннями на мовах для доведення теорем.

0.2 Формалізована постановка задачі

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. Також автор зазирнув у спектр мов, які доречно використовувати як мови нижнього рівня (системне програмування) для програмування середовища виконання.

Після 5 років цього дослідження, виконавши у вигляді вправ декілька імплементацій мов, було прийнято рішення формально оформити всю роботу згідно академічних нормативів. Тому в цій секції дається формальна постановка задачі, яка складається з опису об'єкту та предмету дослідження, мети, цілей та завдань. Дається головна мотивація та аналіз результатів.

0.2.1 Наукова новизна

Починаючи з перших пруверів 60-х років таких як AUTOMATH, пізніших систем Жирара System F та сучасні фібраційні системи з залежними типами мови програмування та системи типів пройшли значний шлях. Сучасні теорії типів присвячені формалізації вищої математики, гомотопічної теорії, симпліціальної геометрії. А системи програмування абсорбували теорії паральних процесів Мілнера та формалізацію лінійних типів які маються застосування також до квантових обчислень та формалізації теоретичної фізики. Такий сучасний стан наукової розробки теми.

Дана робота ґрунтується на гомотопічній теорії типів для потреб математичних мовних засобів, а всі проміжні системи типів, такі як системи Жирара — для виробництва. Також, ця робота розказує про сучасні середовища виконання, які здатні працювати без операційних систем та реазізуються свої систем розподілу часу (планувальники).

Інновація роботи полягає в побудові унікальної замкненої системи яка складається з: 1) системного програмного забезпечення — модального середовища виконання разом з інтерпретатором написаним на формальній мові, разом з базовою бібліотекою та архітектурою прикладного програмування N2O.DEV; 2) прикладного програмного забезпечення — системи вищих формальних мов, для яких надано моделі, імплементації та базова бібліотека разом з математичними компонентами.

У цій роботі представлені дві конфіурації мовних систем, та три вектора атаки для їх дослідження. Перша атака — це побудова замкненої системи мов для компіляції в середовище виконання віртуальної машини BEAM що входить до складу Erlang/OTP. Друга атака — це побудова власної

віртуальної машини CPS, яка пропонує більш формальну та сучасну модель обчислень. Третя атака — це розробка бібліотеки вищих мов, що могла би компілюватися в BEAM та/або CPS.

Табл. 1: Класифікація мов програмування

Домен	**Мови програмування**
HW	VHDL, Verilog, Clash, Chisel, SystemC, Lava, BSV
ASM	PDP-11, VAX, S/360, M68K, PowerPC, MIPS, SPARC, Super-H Intel, ARM, RISC-V
ALG	C, BCPL, ALGOL, SNOBOL, Simula, Pascal, Oberon, COBOL, PL/1
ML	SML, Alice ML, OCaml, UrWeb, Flow, F#
PURE	HOPE, Miranda, Clean, Charity, Joy, Mercury, Elm, PureScript
F_ω	Scala, Haskell, 1ML, Plutus
MACR	LISP, Scheme, Clojure, Racket, Dylan, LFE, CL Nemerle, Nim, Haxe, Perl, Elixir
OOI	Simula, Smalltalk, Self, REBOL, Io JS, Lua, Ruby, Python, PHP, TS, Java, Kotlin
CMP	C++, Rust, D, Swift, Fortran
SHELL	PowerShell, TCL, SH, CLIPS, BASIC, FORTH
SVC	IDL, SOAP, ASN.1, GRPC
MARK	TeX, PS, XML, SVG, CSS, ROFF, OWL, SGML, RDF, SysML
LOGIC	AUT-68, ACL2, LEGO, ALF, Prolog CPL, Mizar, Dedukti, HOL, Isabelle, Z
ΠΣ	Coq, F*, Lean, NuPRL, ATS, Epigram, Cayenne, Idris, Dhall, Cedile, Kind
HoTT	Menkar, Cubical, yacctt, redtt, RedPRL, Arend, Agda
CHKR	TLA+, Twelf, Promela, CSPM
PAR	Ling, Pony, Erlang, BPMN, Ada, E, Go, Occam, Oz
ARR	Julia, Wolfram, MATHLAB, Octave, Futhark, APL SQL, cg, Clarion, Clipper, QCL, K, MUMPS, Q, R, S, J, O

0.2.2 Об'єкт та предмет дослідження

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

Більш розширено, 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; 5) їх поєднання, побудова формальної системи для уніфікованого середовища, яке поєднує середовище виконання та систему верифікації у єдину систему мов та засобів.

Повна класифікація об'єктів дослідження була зроблена в рамках проекту «Енциклопедія Мов Програмування», де була зроблена спроба надати формальну БНФ-нотацію для тих мов, для яких це не було ще ніколи не зроблено (наприклад для APL-подібної мови K).

Предмет дослідження у широкому сенсі — формальні моделі функціональних мов програмування з формальною семантикою. Предмет дослідження у вузькому сенсі — формальні системи Жирара, лямбда-куб Барендрегта та гомотопічні системи, їх категорні моделі, що містять вичерпні математичні властивості.

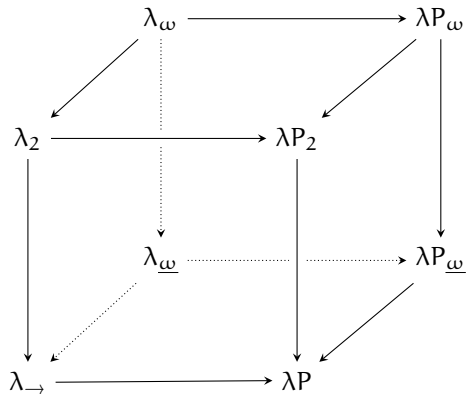
Семантику мов, або властивості мовних категорій, що є предметом дослідження вивчає предмет теорія типів. Предметом та методом дослідження такої системи мов є теорія типів, як сучасний фундамент математики, який стисло та компактно представляє обчислювальне ядро не тільки теорії множин, але і теорії категорій, алгебраїчної топології та дифференціальної геометрії.

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).

Лямбда-куб Барендрегта як карта предмету

Предметом та методом дослідження такої системи мов є теорія типів, як сучасний фундамент математики, який стисло та компактно представляє обчислювальне ядро не тільки теорії множин, але і теорії категорій, алгебраїчної топології та дифференціальної геометрії.

Хенк Барендрегт як автор лямбда кубу, системи формальних моделей які поєднують та класифікують усі лямбда числення в залежності від різного набору чотирьох формул $\star : \star$, $\star : \Box$, $\Box : \Box$, $\Box : \star$. Усі типизовані мови програмування, включаючи PTS (CoC, System P_ω, або чиста система), яка є ядром усіх пруверів, потрапляють у лямбда куб.



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).

0.2.3 Мотивація та мета 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.

Головна мотивація цієї роботи — пошук єдиної мови або мовної системи, що здатна стати уніфікованою мовою, яка пропонує аналогічний спрощений формальний спосіб програмування та доведення теорем.

Кожен інститут чи компанія інвестує в одну певну мову, для зосередження зусиль на одному проекті. Особливість цієї роботи полягає в побудові уніфікованої системи, яка комплексно підходить до вирішення проблеми розширення мовних ядер, та їх обчислювачів. Ця робота пропонує

замкнений фреймворк, який складається з мінімальної системи мов, що покриваються максимальну кількість мовних синтаксисів та семантик.

Крім того, попередні дослідники зосереджувались на побудові системних бібліотек для певного середовища виконання та асоційованих з ним вищих мов (з відповідними біндінгами). На відміну від фокусних досліджень, ця робота пропонує мультимовний підхід, де ми зосереджуємося на побудові моделі, яка буде вбудовуватися з мінімальними зусиллями в основні аглебраїчні мови. У таблиці 0.2 показується можливий ландшафт атаки мовних систем які можна вважати аналогічними підходами до побудови не тільки замкненого життєвого циклу мовного забезпечення, але і системи віртуалізації.

Метою цього дослідження є побудова єдиної системи, яка поєднує формальне середовище виконання та систему верифікації програмного забезпечення з великим спектром мовних засобів, які допомагають вбудувати в себе максимальну кількість існуючих мов. Це прикладне дослідження, яке є сплавом фундаментальної математики та інженерних систем з формальними методами верицікації.

0.2.4 Цілі та завдання дослідження

Головними цілями 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) верифікований 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.

Завданням цього дослідження є імплементація (апробація) специфікації системи мов на різних мовах програмування та типових системах. Для цього проводився аналіз усіх існуючих мов програмування (близько 2000), які детально прокатегоризовані в «Енциклопедії Мов Програмування».

Табл. 2: Ландшафт атаки

Мова	**Чиста мова**	**Імплементація**	**Середовище**	**Віртуалізація**
Henk	PTS	Erlang	BEAM VM	LING/Xen
Per	MLTT	Erlang	BEAM VM	LING/Xen
Henk	PTS	Per	BEAM VM	LING/Xen
Christine	MLTT	Per	BEAM VM	LING/Xen
Alonzo	STLC	OCaml	native	Mirage/Xen
Henk	PTS (CoC)	OCaml	native	Mirage/Xen
Errett	MLTT-72	OCaml	native	Mirage/Xen
Per	MLTT-75	OCaml	native	Mirage/Xen
Giovanni	MLTT-80	OCaml	native	Mirage/Xen
Frank	CoC + CIC	OCaml	native	Mirage/Xen
Christine	CIC + Prop	OCaml	native	Mirage/Xen
Anders	HTS	OCaml	native	Mirage/POSIX
Fabien	$\mathbf{A}^{\mathbf{1}}$-HoTT	OCaml	native	Mirage/POSIX
Dan	∞-HoTT (GAP)	OCaml	native	Mirage/POSIX
Jack	$\Sigma\Omega$-TT	OCaml	native	Mirage/POSIX
Urs	$\flat\sharp$-HoTT	OCaml	native	Mirage/POSIX

Сірим кольором показані мови з екстракцією програм для української віртуальної машини BEAM яка сумісна з байт-кодом Ericsson.

0.2.5 Методи дослідження

Існує багадо підходів для формальної специфікації, верифікації та валідації, усі вони даються у розділі 1, де обгрунтувується 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[8], але і для програмного забезпечення[9]

Табл. 3: Категорні моделі теорій типів

Модель	**Позначення**
Локальні декартово-замкнені категорії Сілі	LCCC
Обширні категорії Гротендіка	CompCat
Природні моделі Еводі	NatMod
Категорії з сімействами Диб'єра	CwF
Категорії з атрибутами	CwA
D-Категорії Картмела	DCat
Контекстуальні системи Воєводського	C-Systems

[8]Близько 10 робіт медалістів премії Філдса грунтуються на категорних методах
[9]Категорні бібліотеки таких мов як Haskell, Scala тощо.

0.2.6 Формальне середовище виконання

В рамках розробленого фреймворку будується архітектура системи доведення теорем та обчислювального середовища (яке складається з верифікованого засобами Rust інтерпретатора та операційної системи, подібно до Erlang, але без надлишкових копіювань), яке побудовано за сучасними стандартами: 1) відсутність системи управління пам'яті у реальному часі (тільки на стадії компіляції); 2) автоматична векторизація за допомогою AVX інструкцій (тензорне ядро); 3) дані ніколи не копіюються; 4) в системі немає очікування та даремного витрачання ресурсів; 5) лінійний лямбда-інтерпретатор, програми якого поміщаються в L1 кеш процесора.

Також додається модель базової бібліотеки середовища виконання для мов Erlang та Hamler, яка пройшла апробацію в підприємствах, державних IT-компаніях та органах виконавчої влади.

0.2.7 Модальна гомотопічна система верифікації

Чиста мова з однією аксомою дається як перша мова системи вищих мов після просто-типізованого лямбда-числення. Далі надається мова класу MLTT та базова бібліотека для екстракції у цільові рантайми BEAM та CPS. І уже після цього, надається гомотопічна мова з відрізком де Моргана, яка сумісна з кубічним верифікатором CCHM, в якому аксіома унівалентності Воєводського має конструктивну інтерпретацію, це наша авторська варіація cubicaltt авторства Андерса Мортберга (2017). Також надається спектральна модель категорій цих мов.

Модальні гомотопічні системи типів та системи типів у зв'язаних топосах є сучасним поглядом на конструктивну математику на шляху до формалізації модальностей, потрібних для інфінітезімальних околів та теорем про нескінченно малі величини, що є вимогами диференціальної геометрії та алгебраїчної топології.

0.2.8 Еквіваріантна супер-гомотопічна система

Додавання модальностей як системи операторів до гомотопічної системи дає змогу дуже елегантно доводити теореми вищої геметрії, диференціальної топології, диференціальної геометрії, супергеометрії. Фінальний аккорд — це побудова формальної M-теорії (спільна робота Урса Шрайбера з Хішамом Саті в Нью-Йоркському Університеті в Абу Дабі). Урс побудував вежу модальностей які вилаштовуються в діагональ спряжень, і чи має ця вежа кінець — відкрите питання в новоспеченій модальній гомотопічній теорії.

Фінальна або термінальна гомотопічна система типів включає в себе примітивні модальності, модальності для диференціальної топології (ретопологізація та топологізація) — шейп модаліті, флет (бемоль) модаліті, шарп (дієз) модаліті, модальності для диференціальної геометрії (Im та Re модальності), фізичні бозонні та ферміонні модальності. Далі система

0.3 Практичні результати

В рамках цього дослідження були досягнуті наступні проміжні практичні результати: 1) проаналізовані та класифіковані всі мови програмування, оформлені у вигляді «Енциклопедії Мов Програмування»[10], що дозволяють виконати проміжні цілі та головне завдання дослідження — створення мінімальної системи мов для побудови уніфікованої системи для програмування та доведення теорем, яка складається з мов середовища виконання та серії вищих мов; 2) розроблений прототип середовища виконання CPS який підтримує імутабельні черги та AMP/SMP планування, містить лінивий інтерпретатор, та підтрмує числення процесів, реалізовано на мові Rust з лінійними типами, для якої існує формальна модель[11] 3) розроблена вища мова програмування Henk з екстрактом в байт-код віртуальної машини BEAM; 4) розроблена базова бібліотека N2O.DEV середовища виконання для віртуальної машини BEAM зі специфікацією в мові Per; 5) розроблена базова бібліотека та бібліотека математичних компонент для вищої модальної гомотопічної мови програмування Anders.

0.3.1 Основні результати

Автор особисто створили моделі та імплементації формального середовища виконання, бібліотеки середовища виконання, декількох верифікаторів та базової бібліотеки як приклади використання та моделювання системи доведення теорем. Автор також розробили курс гомотопічної теорії типів, на якому здійнюється формалізація певних розділів математики (теорія категорій, різні частини алгебраїчної топології та диференціальної геометрії).

Окрім того, створено сайти, присвячені документації по базовій бібліотеці середовища виконання [12], та по бібліотеці системи вищих мов для кубічного верифікатора гомотопічної мови програмування [13]. Також частина моделей розроблений автором попала в апстрім кубічного верифікатора, як приклади використання.

[10] `https://groupoid.github.io/languages`
[11] `https://arxiv.org/pdf/1804.07608.pdf`
[12] `https://n2o.dev/ua`
[13] `https://anders.groupoid.space/lib`

0.3.2 Продукт дослідження

Не зважаючи на нетривіальну структуру результатів, головним продуктом цього дослідження слід вважати загальний підхід описаний в цій роботій, який можна звести до наступних рекомендацій: 1) побудувати лінивий інтерпретатор CPS нетипизованого лямбда числення разом з системою процесів та чергами повідомлень на формальній мові; 2) побудувати мову числення конструкцій PTS на формальній мові з екстрактом у лінийвий інтерпретатор; 3) побудувати мову системи F Жирара або STLC для розробки базової бібліотеки для лінивого інтерпретатора; 4) побудувати базову бібліотеку для лінивого інтерпретатора; 5) побудувати індуктивну мову програмування MLTT; 6) побудувати гомотопічну мову програмування HoTT; 7) побудувати базову бібліотеку для гомотопічної мови; 8) побудувати бібліотеку теорем формальної математики.

Цей підхід закріплюється формальною моделлю даною у розділі 2 та далі розвивається у наступних розділах. Таким чином, можна говорити, що продукт цього дослідження — це формальна специфікація на систему мов та трансформації між ними, яка виявилася цікава не тільки автору дослідження.

0.3.3 Апробація роботи

Апробації: 1) Відбулися виступи на двох міжнародних конференціях: MMCTSE, IAI; 2) Впровадження базової бібліотеки середовища виконання в державних компаніях ПриватБанк, ІНФОТЕХ та органах влади МВС; 3) Впровадження прототипу середовища виконання описаного в роботі, який ліг в основу комерційного продукту.

0.3.4 Аналіз результатів

Ретельне заглиблення в теорію верифікації призвело до компактифікації та покращення бібліотеки середовища виконання. Після апробації роботи автор пересвідчився в правильності вибраної стратегії.

0.4 Структура роботи та публікації

Якшо коротко суть роботи зводиться до побудови системи, яка складається з: 1) середовища виконання; 2) формального інтерпретатора; 3) 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

У розділі 2 розглядається математична модель формальної системи, яка умовно розділяється на систему мов для системного програмування (спектр мов середовища виконання) та систему мов для прикладного програмування і програмування вищих логік (спектр вищих мов). Час дослідження цього направлення припав на 2017-2018 роки.

Формальне середовище виконання

У розділі 3 розказується про повний стек формального програмного забезпечення від віртуальної машини, байт-код інтерпретатора, середовища виконання та планування процесів. Час дослідження цього направлення припав на 2016-2017 роки.

У розділі 4 описується базова бібліотека системи середовища виконання, написана на мові нижчого рівня Erlang. Час дослідження цього направлення припав на 2013-2021 роки.

Система верифікації

У розділі 5 подається опис системи вищих формальних мов для доведення теорем до кубічної теорії та гомотопічної системи типів. Час дослідження цього направлення припав на 2021-2023 роки.

У розділі 6 описується базова бібліотека системи формальних мов, індуктивна та гомотопічні системи типів для кубічних тайпчекерів. Час дослідження цього направлення припав на 2018-2021 роки.

Математичні компоненти

У розділі 7 пропонується ряд математичних моделей та теорій з використанням базової бібліотеки розділу 3 та мови гомотопічної системи типів. Час дослідження цього направлення припав на 2018-2021 роки.

Додаткові матеріали

У додатках надаються приклади іншого використання формальних мов та моделей, зокрема для мінімальної формальної мови, побудованої в рамках дисертації, та мови програмування Coq. А також дається приклад використання гомотопічної мови для формальної філософії.

0.5 Подяка

У вступі хотів би висловити подяку: 1) своїм тибетським гуру без яких ця робота була би неможливою; 2) Андерсу Мортбергу та Хенку Барендрегту; 3) двом вчителям старших класів: геометру Панькову та алгебраїсту Конету; 4) двом вчителям в університеті: Клименко та Таран; 5) усім контрібюторам N2O.DEV[14] та Groupoid Infinity[15]; 6) батькам; 7) іншим вчителям; 8) усім хоробрим людям з почуттям гідності, справжнім українцям.

[14] `https://n2o.dev`
[15] `https://groupoid.space`

Розділ 1

Формальна верифікація

Присвячується вчителям
нідерландської школи

Ніколасу де Брейну,
Хенку Барендрегту
та Барту Якобсу

Перший розділ закладає теоретичний фундамент, оглядаючи сучасний стан формальної верифікації та визначаючи місце пропонованої уніфікованої системи формальних мов у цій галузі.

Розглядаються ключові концепції верифікації та валідації, формальні специфікації, методи верифікації, а також середовища виконання, зосереджуючись на їхньому зв'язку з нашою метою: інтеграцією програмування, доведення теорем і формальної філософії. Розділ завершується аналізом мовних елементів, кластеризованих за теорією типів Мартіна-Льофа, що слугують основою для подальших розробок.

Вступне слово

Формальна верифікація як дисципліна бере початок із робіт Ніколаса де Брейна, чия система AUTOMATH (1967) стала першим механізмом комп'ютеризованого доведення теорем у фіброваній моделі. Хенк Барендрегт розвинув цю ідею, створивши лямбда-куб — класифікацію типових систем, де найвища вершина (Calculus of Constructions, CoC) уможливила вищі формальні мови.

У дисертації ми використовуємо CoC як базис для теоретичного ядра, доповнюючи його вищими мовами для поглинання машинною верифікацією усієї доступної математики. Система окрім вищих мов містить обчислювальне середовище, яке побудовано на інтерпретаторах нетипізованого лямбда числення і числення процесів, а також на мовах

системи Жирара для програмування цього середовища.

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 функціональних 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); 2) Модельно-орієнтований 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).

1.2.1 Програмне забезпечення

Найбільш стандартизована та прийнята в області формальної верификації — це нотація Z[3] (Spivey, 1992), приклад модельно-орієнтованої мови. Назавана мова на честь Ернеста Цермело, роботи якого мали вплив на фундамент математики та аксіоматику теорії множин. Саме теорія

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

множин, та логіка предикатів першого порядку є теорією мови Z. Тут також заслуговують уваги сесійні 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.

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). GAP (Joachim Neubüser, RWTH Aachen, Kaiserslautern).

1.3 Формальні методи верифікації

Можна виділити три підходи до верифікації: 1) спеціалізовані верифікатори моделей, або системи моделювання; 2) алгебраїчні мови для синтетичних моделей і глибокого вбудовування; 3) системи автоматичного доведення теорем і синтезу програм (теорем).

1.3.1 Спеціалізовані системи моделювання

Перший застосовується де вже є певна програма написана на конкретній 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 підходів: 1) VST (CompCert, сертифікація C програм); 2) NuPRL (Cornell University, розподілені системи, залежні типи); 3) TLA+ (Microsoft Research, Леслі Лампорт); 4) Twelf (для верифікації

[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.

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 основної мови перевіряє модель закодовану у ній же. Можливо моделювання логік вищого порядку, лінійних логік, модальних логік, категорних 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 Університету Каргені-Мелона; 4) F* – окремий проект Microsoft Research.

Зараз другий підхід доповнився гомотопічними мовами, де верифікація відбувається з використанням гомотопічної логіки.

Приклади гомотопічних систем: 1) cubicaltt — [8]CCHM імплементація авторства Андерса Мортберга кубічної теорії типів Сімона 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; 6) Anders — кубічне розширення MLTT-80 з двома видами всесвітів та Im модальністю.

1.3.3 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 мови як Erlang,

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

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

JavaScript, Python. До цих пір нетипізоване лямбда числення є одною з мов у яку робиться конвертація доведених программ (екстракція).

Історіографія фібраційних математичних пруверів бере свій початок з Нідерландів. Перший математичний прувер AUTOMATH (і його модифікації AUT-68 та AUT-QE), який був написаний для комп'ютерів розроблявся під керівництвом де Брейна у 1967 році. У цьому прувері був квантор загальності та лямбда функція, таким чином, це був перший прувер, побудований на засадах ізоморфізма Каррі-Говарда-Ламбека. В рамках проєкту AXIO/1 була розроблена мова Henk на мовах Erlang та OCaml.

ML/LCF або метамова і логіка обчислювальних функцій були наступним кроком до осягнення фундаментальної мови простору, тут вперше з'явилися алгебраїчні 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[10], Японія) та Робіна Кокета (Charity[11], 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 можна умовно розділити на два класи: 1) інтерпретатори нетипізованого або просто типізованого (рідше з більш потужними системами типів), лямбда числення з можливими JIT оптимізаціями; 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 можна віднести 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. 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.

[10] `https://web.sfc.keio.ac.jp/~hagino/thesis.pdf`
[11] `https://github.com/devaspot/charity`

Нійбільш цікаві цільові платформи для виконання программ які побудовані на основі формальних доведень для нас є OCaml (тому, що це основна мова екстракту для промислової системи доведення теорем Coq), Rust (тому, що рантайм може бути написаний без використання сміттєзбірника), Erlang (тому, що підтримує неблоковану семантику π-числення) та Pony (тому, що семантика його π-числення побудована на імутабельних чергах та CAS-курсорах).

1.4.1 Формальні інтерпретатори та ОС

Перший прототип, рантайм $\mathsf{O}_{\mathsf{CPS}}$ – лінивий векторизований інтерпретатор (підтримка SSE/AVX інструкцій) та система управління ресурсами з планувальником лінивих програм та системою черг і CAS курсорів у якості моделі π-числення. Розглядалося також використання ядра L4 на мові C, верифікованого за допомогою HOL/Isabell, у якості базової операційної системи. В рамках проєкту AXIO/1 назва цієї мови — Bob.

Наступна версія $\mathsf{O}_{\mathsf{CPS}}$ або подальші дослідження верифікованих інтерпретаторів будуть базуватися на результатах таких досліджень як CoqASM[12] та Verified LISP Interpreter[13]. В рамках проєкту AXIO/1 назва цієї мови — Joe.

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 CPS. Ця робота представлена у вигляді PTS тайпчекера Henk, який вистує 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).

[12] `http://nickbenton.name/coqasm.pdf`
[13] `https://www.cl.cam.ac.uk/~mom22/tphols09-lisp.pdf`

1.5 Висновки

Як результат цього розділу, було досліджено: 1) усі системи доведення теорем; 2) формальні мови програмування; 3) системи верифікації; 4) 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: 5) O_λ — нетипизоване λ-числення Чорча; 6) O_π — числення процесів, CCS, CSP або π-числення Мілнера; 7) O_μ — тензорне числення та векторизація (MatLab, Julia, kx, J); 8) O_Π — числення конструкцій (функціональна повнота); 9) O_Σ — числення контекстів (контекстуальна повнота); 10) $\mathsf{O}_=$ — теорія типів Мартіна-Льофа (логіка); 11) $\mathsf{O}_\mathcal{W}$ — числення індуктивних конструкцій (матіндукція); 12) O_I — гомотопічна система типів (формальна математика).

Розділ 2

Спектральна категорія мов формальної верифікації

Присвячується автору першої дисертації з гомотопічної теорії типів

Майклу Уорену

Другий розділ описує розвиток концептуальної моделі системи доведення теорем на основі спектральної моноїдальної категорі з гомотопічною структурю як сукупності:

1) категорій, які розкривають семантику конкретної теорії типів (мови програмування) як синтаксису, або є її метатеорією; 2) моноїдальні теорії для маніпуляції мовними синтаксисами та програмами (мовні категорії); 3) гомотопічних мовних синтаксисів, як вищих індуктивних типів (CW-комплексів) теорії гомотопій; 4) визначення когомологій, як набору трансформацій між мовам, що зберігають певні інваріанти (виразність, обчислювальну семантику); 5) використання конструкції Гротендіка для аналізу стабільних властивостей мов; 6) загальний фреймворк спектральної категорії.

Ця концептуальна модель є сукупністю категорій, що пропонує фреймворк або стандарт на розробку та формалізацію формальних мов. Усі мови програмування та їх моделі представлені за допомогою цього фреймворка, пропонують сучасну математичну метатеоретичну основу для мов програмування (імплементації конкретних теорій типів).

В наступних розділах 3 та 5 присвячених теоріям типів для програмування та доведення теорем буде йтися виключно про теоретико-типові моделі, тобто мови програмування побудовані на конкретних синтаксисах які представлені індуктивними типами.

Цей розділ буде включати набір моделей, які використовуються для

дослідження категорної семантики типових систем. Теоретичні засади цього фібраційного категорного фундаменту були закладені Александром Гротендіком в 1971 році.

2.1 Категорні моделі мов

Категорні засади залежної теорії були побудовані Робертом Сілі. [1]. Семантика залежної теорії поліморфного лямбда-числення вищих порядків вивчається локальними декартово замкненими категоріями (ДЗК) — категоріями з кінечними лімітами, де слайс-категорії по будь-якому об'єкту є декартово-замкненими.

Перші спрови побудови повної формальної категорної семантики теорії типів були дані Томасом Страйхером в 1991 році [2]. Пітер Диб'єр запропонував категорну модель з сімействами (Categories with Families, CwF) в 1995 році [3], а також Мартін Хофман показав, що ці категорії ізоморфні категоріям з атрибутами (Categoties with Attributes, CwA) в 1997 році [4]. Стів Еводі запропонував трохи модифіковану категорну досніпову модель теорії типів, яку назвав природньою моделлю теорії типів [5].

2.1.1 Вступне слово

Головною проблемою представлення категорної семантики теорії типів, крім доведення коректності та повноти, є теорема про ініціальність, яка говорить, що ініціальними об'єктами в категорії моделей системи типів повинна бути модель термів. Таким чином, теорії типів пакуються в категорні моделі, які містять усі необхідні теореми пов'язані з когерентністю (яка була вирішена в категоріях розширень[6]) усіх композицій, рівняннями та властивостями нормалізації та обчислень, канонічного представлення, тощо. Повна формальна ініціальна модель MLTT була представлена на Agda лише в 2020 році за допомогою алгебраїчної теорії категорій [1].

Генеалогічна лінія передачі 2-категорного моделювання бере початок з шостої глави SGA-1 лекцій Александра Гротендіка 1971 [7]. В 1984 році Сілі [1] опублікував роботу по локальним ДЗК, однак в такій моделі кожна стрілка є типом, що недостатньо деталізує модель. Інший підхід запропонований Джоном Картмелом [8] полягав в створенні контекстуальних категорій, що дозволяв більш простіше і безпосередніше виражати природу залежних термів. Ці дві моделі були уніфіковані Томасом Ерхардом в 1988 році [9] більш абстрактною моделлю, яка була глибшою і тим ближчою до оригіналу (SGA-1), в цій моделі були представлені категорії розширень. Категорії розширень Гротендіка також можна знайти в роботах Барта Якобса[10][11].

[1] `https://github.com/guillaumebrunerie/initiality` – доведення кон'юнктури ініціальності MLTT

Модельні категорії Деніела Квіллена [12] пропонують ще вищий спосіб представлення, де розкривається глибинна структура просторів за допомогою слабких систем факторизації. Для сучасних математичних мов програмування (кубічна теорія типів) побудовані модельні категорії Квіллена, однак кон'юнктура ініціальності залишається відкритою проблемою (станом на 2023 рік).

Подальші дослідження Террі Кокана категорних моделей теорії типів виходять на давню мрію Гротендіка про стекову модель, де замість досніпа зі значенням в категорії множин, береться функтор зі значеннями в категорії вищих групоїдів [13].

2.1.2 Кон'юнктура ініціальності

Кон'юнктура ініціальної в теорії типів стверджує, що модель термів (всі терми і всі типи) певної системи типів повинна бути ініціальним об'єктом в категорії моделей цієї системи типів. Ініціальність валідує формальний перехід від теорії категорії до системи типів, таким чином, що синтаксичні пруфтерми системи типів в точності відповідають теоремам в категорії, які інтерпретують ці системи типів.

Вперше ініціальність для чистих систем (Pure Type Systems), які складаються з одного П-типу, була дана Томасом Страйхером. З тих пір Ініціальний для більш складних типових систем, як то Martin-Löf Type Theory (MLTT), вважалася досягнута, вважаючи механічне продовження техніки категорного формалізації для інших типів (Π, Σ, $=$, $+$, $\bot$, $\top$, N, U_i, El). Хоча багато дослідників, починаючи з Володимира Воєводського і його серії статей присвячених важливості механістичної формалізації кон'юнктури ініціальної в 2015-2017 рр, займалися дослідженнями ініціальності, але лише в 2020 році Гійом Брунері та Пітер Люмсдейн спільно з Менно де Боєр і Андерсом Мёртбергом представили формальну модель ініціальної MLTT на Agda.

2.1.3 Концепти та Категорії

Концепти Фреге були першою спробою фібраційної формалізації основи для математичних тверджень, яка складається з розшарування (за допомогою якого моделюється квантор узагальнення) та тотального простору (за допомогою якого моделюється квантор існування).

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

Definition 2. (Система). Визначимо систему як сукупність об'єктів $\mathtt{Ob} : \mathsf{U}$ та зв'язків між ними $\mathtt{Hom} : \mathtt{Ob} \to \mathtt{Ob} \to \mathsf{U}$ які називаються морфізмами.

Definition 3. (Докатегорія). Категорія – це система яка має дві операції: 1) для кожного об'єкта системи існує одиничний морфізм $\mathtt{id}$; 2) для кожних двох морфізмів системи існує операція їх композиції $\circ$, які повинні мати три властивості: 1) ліва композиція з одниничними морфізмами; 2) права композиція з одиничними морфізмами; 3) асоціативність композиції. Якщо дві операції мають тільки третю властивість асоціативність то кажуть про напівкатегорії.

Definition 4. (Категорія). Категорія це докатегорія, така що для довільного $\mathsf{A} : \mathtt{Ob}(\mathtt{c})$ тип $\mathtt{isContr}(\Sigma(\mathsf{B} : \mathtt{Ob}(\mathsf{C})), \mathsf{A} = \mathsf{B})$ населений.

Definition 5. (Мала категорія). Якщо морфізми категорії утворють множину то така категорія називається малою.

Definition 6. (Концептуальна модель). Концептуальна модель визначається як категорія, об'єкти якої індексовані певною множиною, або залежні від параметра.

2.1.4 Відповідність між категорними моделями

Для того аби показати сучасну досніпову категорну модель теорії типів, дамо которкий опис теорій, які лягли в основу категорного моделювання теорії типів.

Табл. 2.1: Категорні моделі теорій типів

Категорна модель	**Позначення**
Локальні декартово-замкнені категорії Сілі	LCCC
Обширні категорії Гротендіка	CompCat
Природні моделі Еводі	NatMod
Категорії з сімействами Диб'єра	CwF
Категорії з атрибутами	CwA
D-Категорії Картмела	DCat
Контекстуальні системи Воєводського	C-Systems

2.1.5 Обширні категорії Гротендіка

Definition 7. (Вертикальний морфізм). Нехай $\mathtt{p} : \mathtt{E} \to \mathtt{B}$ — функтор. Морфізм $\mathtt{f} : \mathtt{Y} \to \mathtt{X}$ називається вертикальним, якщо $\mathtt{p}(\mathtt{f})$ є одиничним морфізмом в $\mathtt{B}$: $\mathtt{p}(\mathtt{f}) = \mathtt{id}_{\mathtt{B}}$.

Definition 8. (Декартовий морфізм). Нехай $\mathtt{p} : \mathtt{E} \to \mathtt{B}$ — функтор, морфізм $\mathtt{f} : \mathtt{Y} \to \mathtt{X}$ категорії $\mathtt{E}$ називається дакартовим над $\mathtt{t} = \mathtt{p}(\mathtt{f})$ якщо для кожного морфізма $\mathtt{u} : \mathtt{Z} \to \mathtt{Y}$ такого що $\mathtt{p}(\mathtt{f}) = \mathtt{p}(\mathtt{u})$ існує унікальна вертикальна стрілка $\mathtt{a} : \mathtt{Z} \to \mathtt{Y}$ така що $\mathtt{f} \circ \mathtt{a} = \mathtt{u}$.

Definition 9. (Гіпердекартовий морфізм, Бенабу). Нехай $\mathtt{p} : \mathtt{E} \to \mathtt{B}$ — функтор, морфізм $\mathtt{f} : \mathtt{Y} \to \mathtt{X}$ категорії $\mathtt{E}$ називається гіпердакартовим над $\mathtt{t} = \mathtt{p}(\mathtt{f})$ якщо для кожного морфізма $\mathtt{u} : \mathtt{K} \to \mathtt{J}$ категорії $\mathtt{B}$ та кожного морфізма $\mathtt{v} : \mathtt{Z} \to \mathtt{X}$ категорії $\mathtt{E}$, таких, що $\mathtt{p}(\mathtt{v}) = \mathtt{t} \circ \mathtt{u}$ існує унікальний морфізм $\mathtt{w} : \mathtt{Z} \to \mathtt{Y}$ категорії $\mathtt{E}$, такий, що $\mathtt{v} = \mathtt{f} \circ \mathtt{w}$ та $\mathtt{p}(\mathtt{w}) = \mathtt{u}$.

Definition 10. (Категорія стрілок). Категорія стрілок $\mathtt{B}^{\to}$ категорії $\mathtt{B}$ є категорією, у якої об'єкти це стрілки категорії $\mathtt{a} : \mathtt{Ob}^{\to}_{\mathtt{B}} = \mathtt{Hom}_{\mathtt{B}}(\mathtt{x}, \mathtt{y})$, а морфізми — пари стрілок $\mathtt{Hom}^{\to}_{\mathtt{B}} = [\mathtt{f} : \mathtt{Hom}_{\mathtt{B}}, \mathtt{g} : \mathtt{Hom}_{\mathtt{B}}]$ з категорії $\mathtt{B}$, які комутують:

$$\begin{array}{ccc} \mathtt{x} & \xrightarrow{\mathtt{f}} & \mathtt{f}(\mathtt{x}) \\ {\scriptstyle \mathtt{a}}\downarrow & & \downarrow{\scriptstyle \mathtt{f}(\mathtt{a})} \\ \mathtt{y} & \xrightarrow{\mathtt{g}} & \mathtt{f}(\mathtt{y}) \end{array}$$

Definition 11. (Розшаруванням Гротендіка). Функтор $\mathtt{p} : \mathtt{E} \to \mathtt{B}$ називається розшарованою категорією над $\mathtt{B}$ (або розшаруванням Гротендіка), якщо для кожного морфізма $\mathtt{u} : \mathtt{J} \to \mathtt{I}$ в категорії $\mathtt{B}$ та об'єкта $\mathtt{X} \in \mathtt{p}(\mathtt{I})$ в категорії $\mathtt{B}$, існує декартовий морфізм $\mathtt{f} : \mathtt{Y} \to \mathtt{X}$ над $\mathtt{u}$ який називається декартовим підйомом $\mathtt{X}$ над $\mathtt{u}$.

Definition 12. (Розщеплене розшарування). Розшарування Гротендіка називається розщепленим або функторіальним досніпом, якщо $\mathtt{p}^{-1}$ може

бути продовжений до функтора $\mathsf{B}^{\mathsf{op}} \to \mathtt{Cat}$ в точній індексованій категорії. Категорія, об'єкти якої є розщеплені розшарування позначається як функторіальний досніп $\mathtt{Psh}(\mathsf{B}) = [\mathsf{B}^{\mathsf{op}}, \mathtt{Cat}]$.

Definition 13. (Розшарований функтор). Нехай $\mathsf{p} : \mathsf{X} \to \mathsf{B}$ та $\mathsf{q} : \mathsf{Y} \to \mathsf{B}$ — розшарування Гротендіка зі спільною базою B, Функтор $\mathsf{F} : \mathsf{X} \to \mathsf{Y}$ називається декартовим (або розшарованим функтором), якщо: 1) $\mathsf{q} \circ \mathsf{F} = \mathsf{P}$; 2) для кожного декартового морфізма x з X відносно p, $\mathsf{F}(\mathsf{x})$ — декартовий морфізм відносно q.

Definition 14. (Розшароване природнє перетворення). Нехай $\mathsf{p} : \mathsf{E} \to \mathsf{B}$ та $\mathsf{q} : \mathsf{D} \to \mathsf{A}$ це два розшарування Гротендіка зі спільною базою B, тоді категорія $\mathtt{Fib}_{\mathsf{B}}(\mathsf{p}, \mathsf{q})$ (підкатегорія $\mathtt{Cat}/\mathsf{B}$) визначається так, що об'єкти це розшаровані функтори $\mathsf{p} \to \mathsf{q}$, а морфізми це пари функторів $(\mathsf{H} : \mathsf{E} \to \mathsf{D}, \mathsf{K} : \mathsf{B} \to \mathsf{A})$ такі, що для довільного декартового морфізма f відносно p слідує, що $\mathsf{H}(\mathsf{f})$ декартовий відносно q.

Definition 15. (Обширна категорія). Функтор $\mathsf{p} : \mathsf{E} \to \mathsf{B}^{\to}$ називається обширною категорією, якщо: 1) $\mathtt{cod} \circ \mathsf{p} : \mathsf{E} \to \mathsf{B}$ є декартовим функтором; 2) для кожного декартового функтора $\mathsf{f} \in \mathsf{E}$ значення функтора p в точці f є пулбеком в B.

Definition 16. (Конструкція Гротендіка). Ізоморфізм між 2-категоріями $\mathtt{Psh}(\mathsf{B})$ та $\mathtt{Fib}(\mathsf{B})$ називається конструкцією Гротендіка.

$$\int : \mathtt{Psh}(\mathsf{B}) \xrightarrow{\cong} \mathtt{Fib}(\mathsf{B})$$

2.1.6 Локальні декартово-замкнені категорії Сілі

Локальні декартово-замкнені категорії (ДЗК) та їх зв'язок з теорією типів були представлені Сілі[1]. Внутрішньою мовою локальних ДЗК є мова програмування з залежними типами Π та Σ, що становить основу сучасних фібраційних пруверів. Доведення, що декартово замкнена категорія містить STLC надано в розділі 7 математичних компонент в рамках топосо-теоретичної моделі конструктивної теорії множин.

2.1.7 Категорії з сімействами Диб'єра

Узагальнена алгебраїчна теорія Пітера Диб'єра [3][14][15].

Definition 17. (Категорія з сімействами). Категорія C, об'єкти якої Ob_{C} це простори залежних функцій $\Pi(A, B)$, а морфізми $\mathsf{Hom}_{\mathsf{C}}(\Pi(A, B), \Pi(A', B')$ пари функцій $[f : A \to A', g(x : A) : B(x) \to B'(f(x))$.

Definition 18. (Категорія контекстів). Категорія, об'єкти якої є усі можливі контексти, а морфізми усі можливі підстановки.

```
def CwF : U := Σ (C: precategory) (T: catfunctor C Fam)
    (context: isContext C) (terminal: isTerminal C), isComprehension C T
```

2.1.8 Природні моделі Еводі

Сучасна категорна досніпова модель теорії типів, яка була представлена Стівом Еводі та активно розроблюється в університеті Карнегі-Мелона.

Definition 19. (Досніп). Досніп на категорії C визначається як функтор $\mathsf{F} : \mathsf{C}^{\mathsf{op}} \to \mathbf{Set}$ з оберненої до C категорії в категорію множин $\mathbf{Set}$.

Definition 20. (Природна модель). Природна модель складається з: 1) категорії C; 2) виділеного термінального об'єкту $\mathsf{t} \in \mathsf{C}$; 3) досніпів $\mathsf{Ty}, \mathsf{Tm} : \mathsf{C}^{\mathsf{op}} \to \mathbf{Set}$; 4) природнього перетворення $\mathsf{p} : \mathsf{Tm} \to \mathsf{Ty}$.

```
def naturalModel : U := Σ (C  : precategory) (_  : isCategory C)
    (t : terminal C) (Tm : carrier C) (Ty : carrier C)
    (p : hom C VT V), Π (f : homTo C V), hasPullback C (Tm, f, Ty, p)
```

Володимир Воєводський запропонував свою категорну модель, яку назвав $\mathsf{C}-$системами. В бібліотеці математичних компнент розділу 7 представлене доведення ізоморфізму $\mathsf{C}-$систем Воєводсього природним моделям Еводі.

Definition 21. (Репрезентативні природні перетворення). Для двох досніпів $\mathsf{P}, \mathsf{Q} : \mathsf{C}^{\mathsf{op}} \to \mathbf{Set}$ та природнього перетворення $\alpha : \mathsf{Q} \to \mathsf{P}$, α називається репрезентативним якщо для всіх $\mathsf{Ob}(\mathsf{C})$ та $\chi : \mathsf{Ob}(\mathsf{C})$ існує $\mathsf{p}_\chi : \mathsf{D} \to \mathsf{C}$ та $\mathsf{y} : \mathsf{Q}(\mathsf{D})$, такий що цей квадрат комутує:

$$\begin{array}{ccc} \mathsf{y}(\mathsf{D}) & \xrightarrow{\mathsf{y}} & \mathsf{Q} \\ \downarrow{\scriptstyle \mathsf{y}\mathsf{p}_\chi} & & \downarrow{\scriptstyle \alpha} \\ \mathsf{y}(\mathsf{C}) & \xrightarrow{\chi} & \mathsf{P} \end{array}$$

Definition 22. (Послабляючий морфізм). Складається з: 1) функтора природніх моделей $\mathsf{F} : \mathsf{C} \to \mathsf{D}$, 2) природне перетворення $\phi_{\mathsf{Ty}} : \mathsf{F}_!\mathsf{Ty}_\mathsf{C} \to \mathsf{Ty}_\mathsf{D}$, 3) природне перетворення $\phi_{\mathsf{Tm}} : \mathsf{F}_!\mathsf{Tm}_\mathsf{C} \to \mathsf{Tm}_\mathsf{D}$, такими що наступа діаграма комутує:

$$\begin{array}{ccc} \mathsf{F}_!\mathsf{Tm}(\mathsf{C}) & \xrightarrow{\phi_{\mathsf{Tm}}} & \mathsf{Tm}(\mathsf{D}) \\ \downarrow{\scriptstyle \mathsf{F}_!\mathsf{p}(\mathsf{C})} & & \downarrow{\scriptstyle \mathsf{p}(\mathsf{D})} \\ \mathsf{F}_!\mathsf{Ty}(\mathsf{C}) & \xrightarrow{\phi_{\mathsf{Ty}}} & \mathsf{Ty}(\mathsf{D}) \end{array}$$

.

Тут $\mathsf{F}_! : \mathbf{Set}^{\mathsf{C}^{\mathsf{op}}} \to \mathbf{Set}^{\mathsf{D}^{\mathsf{op}}}$ є лівим розширенням Кана.

2.1.9 Модельні категорії Квіллена

Дисертація Деніела Квіллена була присвячена диференціальним рівнянням, але відразу після цього він перевівся в MIT і почав працювати в алгебраїчній топології, під впливом Дена Кана. Через три роки він видає Шпрінгеровскі лекції з математики "Гомотопічна алгебра яка назавжди транформувала алегбраїчну топологію від вивчення топологічних просторів з точністю до гомотопий до загального інструменту, що застосовується в інших областях математики.

Модельні категорії вперше були успішно застосовані Воєводським для доказу кон'юнктури Мілнора (для 2) і потім мотивной кон'юнктури Блоха-Като (для n). Для доказу для 2 була побудована зручна стаблільная гомотопічні категорія узагальнених схем. Інфініті категорії Джояля, досить добре досліджені Лур'є є прямим узагальненням модельних категорій.

Цікавою властивістю модельних категорій є те, що дуальні до них категорії перевертають розшарування і корозшарування, таким природнім чином реалізують дуальність Екмана-Хілтона. Розшарування і корозшарування пов'язані, тому взаємовизначені. Корозшарування є морфізмами, що мають властивість лівого гомотопічні підйому по відношенню до ациклічним розшарування і розшарування є морфізмами, що мають властивість правого гомотопічні підйому по відношенню до ациклічних корозрашувань.

2.2 Спектральна категорія формальних мов

Категорії, об'єкти яких є мови програмування, або точніше їх синтаксиси (ініціальні об'єкти), а морфізми — транформаціями цих синтаксичних дерев (верифікаторами, компіляторами, екстраторами) є об'єктом дослідження концептуальної ситсеми мов.

В той час, як категорні моделі теорії типів працюють з контекстуальними категоріями та двома досніпами Tm та Ty які моделюють типи та терми в категорії множин, мовні категорії призначені для моделювання різних теорій типів та різних відповідних досніпів, а також перетворень між ними.

Definition 23. (Синтаксичне дерево). Синтаксичне дерево — це індуктивний тип або дерево Бома, контруктори якого відповідають одному з 4 типів правил в теорії типів, як правило використовуються три правила: правило формації, інтро-правила та елімінатор.

Definition 24. (Вище синтаксичне дерево). Синтаксичне дерево в яке додано β та η правила називається вищим синтаксичним деревом.

Definition 25. (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, $\mathsf{O}_=$. Приклади мовних категорій: $\mathsf{O}_{\mathsf{PTS}}$ (Henk), $\mathsf{O}_{\mathsf{MLTT}-80}$ (Per), $\mathsf{O}_{\mathsf{HTS}}$ (Anders).

Definition 26. (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.

Definition 27. (Послідовність синтаксичних дерев). Кожна послідовність синтаксичних дерев

$$\mathsf{O}_\Pi \to \mathsf{O}_\Sigma \to \mathsf{O}_= \to \mathsf{O}_W \to \mathsf{O}_\mathrm{I}. \tag{2.1}$$

генерує відповідну послідовність мов програмування

$$\begin{aligned} \mathsf{O}_{\mathsf{PTS}}(\mathsf{O}_\Pi) \to \mathsf{O}_{\mathsf{MLTT}-72}(\mathsf{O}_\Pi, \mathsf{O}_\Sigma) \to \mathsf{O}_{\mathsf{MLTT}-75}(.., \mathsf{O}_\Sigma, \mathsf{O}_=) \to \\ \to \mathsf{O}_{\mathsf{MLTT}-80}(..., \mathsf{O}_=, \mathsf{O}_W) \to \mathsf{O}_{\mathsf{HTS}}(..., \mathsf{O}_W, \mathsf{O}_\mathrm{I}). \end{aligned} \tag{2.2}$$

наступним чином. Кожна мова програмування залежить від синтаксису який її визначає та всіх попредніх синтаксисів мов програмування з послідовності. Перша мова програмування містить тільки перший синтаксис. Розкриті сигнатури мають вигляд: $\mathsf{O}_{\mathsf{PTS}} : \mathsf{O}_\Pi \to \mathsf{U}$,
$\mathsf{O}_{\mathsf{MLTT}-72} : \mathsf{O}_\Pi \to \mathsf{O}_\Sigma \to \mathsf{U}$,
$\mathsf{O}_{\mathsf{MLTT}-75} : \mathsf{O}_\Pi \to \mathsf{O}_\Sigma \to \mathsf{O}_= \to \mathsf{U}$,
$\mathsf{O}_{\mathsf{MLTT}-80} : \mathsf{O}_\Pi \to \mathsf{O}_\Sigma \to \mathsf{O}_= \to \mathsf{O}_W \to \mathsf{U}$,
$\mathsf{O}_{\mathsf{HTS}} : \mathsf{O}_\Pi \to \mathsf{O}_\Sigma \to \mathsf{O}_= \to \mathsf{O}_W \to \mathsf{O}_\mathrm{I} \to \mathsf{U}$.

Табл. 2.2: Аналіз формальних суб-мов як примітивів ядра

Мова	**Застосування**
O_λ	Нетипизоване λ-числення Чорча (інтерпретація)
O_π	Числення процесів, CCS, CSP або π-числення Мілнера
O_μ	Тензорне числення (векторизація)
O_Π	Числення конструкцій PTS (функціональна повнота)
O_Σ	Числення контекстів MLTT-72 (контекстуальна повнота)
$\mathsf{O}_=$	Теорія типів Мартіна-Льофа MLTT-75 (логіка)
$\mathsf{O}_\mathcal{W}$	Індуктивні конструкції MLTT-80 (матіндукція)
O_I	Гомотопічна система типів CCHM (формальна математика)
$\mathsf{O}_\triangleright$	CCHM з обмежувальною рекурсією (теореми про π-числення)
$\mathsf{O}_/$	Система фактор-типів (Lean)
O_H	Мова з оператором Адамара (квантова фізика)
$\mathsf{O}_\dashv$	Модальна HoTT (фізика)

Сірим кольором показаний спектр мовних примітивів ядра концептуальної моделі.

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

Definition 28. (Створення мовної категорії). Мови можна додавати, наприклад $\mathsf{O}_\mathrm{HTS} = \mathsf{O}_{\Pi\Sigma=\mathcal{W}\mathrm{I}}$, 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:

$$\mathsf{O}_\mathrm{HTS} = \mathsf{O}_{\Pi\Sigma=\mathcal{W}\mathrm{I}} : \mathsf{O}_\Pi \to \mathsf{O}_\Sigma \to \mathsf{O}_= \to \mathsf{O}_\mathcal{W} \to \mathsf{O}_\mathrm{I} \to \mathsf{U}. \tag{2.3}$$

Кожне синтаксичне дерево, як правило, містить конструктори та елімінатори певного одиничного типу. Але починаючи з $\mathsf{O}_{\mathrm{MLTT}-80}$ складність типів, які додаються до ядра значно зростає. Таким чином мовні категорії конструюються гранулярно з точністю до включення певного типу в ядро верифікатора.

Definition 29. (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.

Definition 30. (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.

$$\mathsf{O}_\infty : \mathsf{O}_{\mathsf{CPS}} \to \mathsf{O}_{\mathsf{PTS}} \to \mathsf{O}_{\mathsf{MLTT-80}} \to \mathsf{O}_{\mathsf{HTS}} \to ... \tag{2.4}$$

Definition 31. (Фільтри). Фільтр у мовній категорії — це морфізм $\mathsf{f} : \mathsf{O}_\mathsf{x} \to \mathsf{O}_\mathsf{y}$, який трансформує мову O_x в мову O_y, через додавання, видалення, перетворення, оцінку чи нормалізацію її властивостей. Множина всіх фільтрів позначається F_O.

Definition 32. (Таксономія фільтрів). Фільтри $\mathcal{F}_\mathcal{L}$ поділяються на підмножини: 1) $\mathcal{F}_{\text{enrich}}$: Фільтри збагачення, $\mathsf{f} : \mathsf{L} \to \mathsf{L}', \mathsf{L}' \supseteq \mathsf{L}$ за інформацією (наприклад, infer). 2) $\mathcal{F}_{\text{simp}}$: Фільтри спрощення, $\mathsf{f} : \mathsf{L} \to \mathsf{L}', \mathsf{L}' \subseteq \mathsf{L}$ за інформацією (наприклад, erase). 3) $\mathcal{F}_{\text{trans}}$: Фільтри трансформації, $\mathsf{f} : \mathsf{L} \to \mathsf{L}', \mathsf{L}' \cong \mathsf{L}$ за виразністю (наприклад, compile). 4) $\mathcal{F}_{\text{eval}}$: Фільтри оцінки, $\mathsf{f} : \mathsf{L} \to \mathsf{O}$, де (O) — граничний об'єкт (наприклад, check). 5) $\mathcal{F}_{\text{norm}}$: Фільтри нормалізації, $\mathsf{f} : \mathsf{L} \to \mathsf{L}'$, де L' — канонічна форма (L) (наприклад, reduce). 6) $\mathcal{F}_{\text{certify}}$: Фільтри сертифікації, $\mathsf{f} : \mathsf{L} \to \mathsf{L}'$, де L' — (наприклад, certify).

Definition 33. (Коконтекстуальна категорія мов). Якщо не виділяти певну послідовність мовного ускладнення та розглядати усі суми усієї певної множини мовних синтаксисів, то ми отримаємо коконтекстуальну 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: $\mathsf{O}_{\mathsf{I}*} \to \mathsf{O}_{\Pi=}$, $\mathsf{O}_\Pi \to \mathsf{O}_{\Pi\Sigma}$, $\mathsf{O}_\Pi \to \mathsf{O}_{\Pi\Sigma}$, $\mathsf{O}_{\Pi*} \to \mathsf{O}_\Pi$.

2.2.1 Структурне представлення моделі

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

Henk = U^n, Π.
Frank = U^n, Π, Ind.
Errett = U^n, Π, Σ, Prop.
Per = U^n, Π, Σ, Path.
Giovanni = U^n, Π, Σ, 0, 1, 2, W, Prop.
Christine = U^n, Π, Σ, Id, Ind, Prop
Anders = U^n, V^n, Π, Σ, 0, 1, 2, W, Path, Prop.
Urs = Anders, $\mathsf{U}_\mathsf{i}^\mathsf{g}$, A × B, $\mathbf{G} \to$ A, $\mathfrak{s}$, $\flat$, $\sharp$, $\Im$, $\bigcirc$.
Dan = Anders, Chain, Cochain, Category, Monoid, Group, Ring, Δ.
Fabien = Anders, k, $\mathbf{A}^1$, $\mathsf{S}^{1,1}$, $\mathsf{L}_{\mathbf{A}^1}$, Susp, $||\_||^\mathsf{n}$, Nisn, $\mathsf{K}^1(\mathbb{Z},\mathfrak{n})$, BGL, MGL.
Jack = Anders, Fib^n, Susp, $\mathsf{Trunc}^\mathsf{n}$, $\mathbf{N}$, $\mathbf{N}_\infty$, Spec, $\pi_\mathfrak{n}\hat{}\mathsf{S}(\mathsf{A})$, $\mathsf{S}^\mathsf{o}[\mathsf{p}]$, Group, A ∪ B, [A, B], $\mathsf{H}^\mathsf{n}(\mathsf{X};\ \mathsf{G})$, G ⊗ H, SS(E, r).

2.2.2 Мінімальна система

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

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

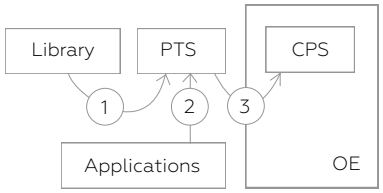


Рис. 2.1: Мінімальна система з чистої мови та інтерпретатора

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.

2.2.3 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).

$$\mathtt{Total} = \begin{cases} \mathtt{Ob} : \{\mathrm{O}_{\mathrm{CPS}}, \mathrm{O}_{\mathrm{PTS}}, \mathrm{O}_{\mathrm{MLTT-75}}, \mathrm{O}_{\mathrm{MLTT-80}}, \mathrm{O}_{\mathrm{HTS}}\} \\ \mathtt{Hom} : \begin{cases} 1, 2 : \not\vdash \to \mathrm{O}_{\mathrm{HTS}}, 3 : \mathrm{O}_{\mathrm{MLTT-75}} \to \mathrm{O}_{\mathrm{MLTT-80}} \\ 4 : \mathrm{O}_{\mathrm{HTS}} \to \mathrm{O}_{\mathrm{MLTT-80}}, 5 : \mathrm{O}_{\mathrm{MLTT-80}} \to \mathrm{O}_{\mathrm{PTS}}, 6 : \mathrm{O}_{\mathrm{PTS}} \to \mathrm{O}_{\mathrm{CPS}} \end{cases} \end{cases} \tag{2.6}$$

За допомогою мереж Петрі це можна відобразити наступним чином:

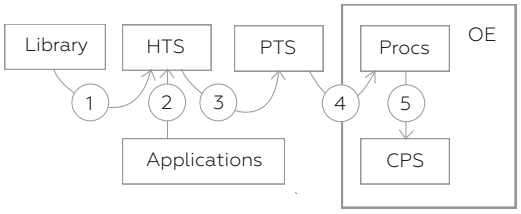


Рис. 2.2: Кубічна та чиста системи типів та середовище виконання

2.2.4 Категорія середовища виконання CPS

Definition 34. (Категорія середовища виконнання $\mathsf{O}_{\mathsf{CPS}}$).

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

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

Definition 35. (Синтаксис мовної категорії $\mathsf{O}_{\mathsf{CPS}}$).

```
def CPS : U
 := inductive { lambda (c: Joe CPS)
              | process (m: Bob CPS)
              | tensor (f: Alice CPS)
              }
```

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 $\mathsf{O}_{\mathsf{CPS}}$, тому об'єкти мовних категорій є maybe-типами.

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

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.

Definition 36. (Синтаксичне дерево O_λ). Інтерпретатор визначається своїм трьома конструкторами: номер змінної (індекс де Брейна), лямбда функція та її апплікація:

```
def Joe (cps: U): U
 := inductive { 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.

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

Definition 37. (Синтаксичне дерево O_π).

```
def Bob (lang: U) : U
 := inductive { process (protocol: lang)
              | spawn (cursors: lang) (core: nat) (program: lang)
              | snd (cursor: lang) (data: lang)
              | rcv (cursor: lang)
              | pub (size: nat)
              | sub (cursor: lang)
              }
```

Definition 38. (Синтаксичне дерево O_μ).

```
def Alice (lang: U) : U
 := inductive { Variable (_: Var)
              | Prim (_: Builtin)
              | Star | True | False
              | Int (_: nat) | Float (_: float)
              | Lambda (a: Var) (b: Linear) (c: Exp)
              | App (a b: Exp)
              | Pair (a b: Var) (c d: Exp)
              | Consume (a: Var) (b c: Exp)
              | Gen (a: Var) (b: Exp)
              | Spec (a: Exp) (b: Fraction)
              | Fix (a b: Var) (c d: Linear) (e: Exp)
              | If (a b c: Exp)
              | Let (a: Var) (b c: Exp)
              }

def Linear : U
 := inductive { Empty | Unit | Bool
              | Int | Float
              | Tensor (a: Fraction) (x: Dimension)
              | Pair (a b: Linear) | Fun (a b: Linear)
              | Consume (a: Linear) | All (a: Var) (b: Linear)
              }

def Builtin : U
 := inductive { Intop (a: Arith) | Floatop (a: Arith)    — SIMD types
              | Get | Set | Duplicate | Free             — linearity
              | Transpose | Size                         — matrices
              | Asum | Axpy | Dotp | Rotm | Scal | Amax  — BLAS Level 1
              | Symm | Gemm | Syrk | Posv                — BLAS Level 3
              }

def Fraction  : U := inductive { Z | S (_: Fraction) }
def Dimension : U := inductive { Vector | Matrix | Stream | Table }
def Arith     : U := inductive { Add | Sub | Mul | Div | Eq | Lt | Gt }
```

2.3 Гомотопічні синтаксиси мов програмування

Тут йдеться про мови програмування придатні для доведення теорем, та їх таксономію від найелементарніших (чистої системи з одним типом Π) 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 $\mathbb{N} \to \mathcal{U}$) послідовність мов, кожен елемент якої є мовою програмування, яка не містить синтаксичне дерево вищої мови програмування.

2.3.1 Чиста система типів 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.

Definition 39. (Мовна категорія чистої мови $\mathsf{O}_{\mathsf{PTS}}$).

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

Definition 40. (Синтаксис мовної категорії $\mathsf{O}_{\mathsf{PTS}}$). Чиста мова $\mathsf{O}_{\mathsf{PTS}}$ містить лише синтаксис одного типу, П-типу. Така теорія називається теорією з одним типом, або з однією аксіомою.

```
def PTS : U := inductive { forall (_: Pi PTS) }
def Henk := PTS
```

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

Definition 41. (Синтаксичне дерево O_{Π}).

```
def Pi (lang: U) : U
 := inductive { fibrant (n: nat)
              | variable (x: name) (l: nat)
              | pi (x: name) (l: nat) (f: lang)
              | lambda  (x: name) (l: nat) (f: lang)
              | application (f a: lang)
              }
```

2.3.2 Теорія типів Мартіна-Льофа MLTT-75

Мова теорії типів є сучасною основою всіх пруверів з залежними типами, такими, наприклад, як NuPRL та Agda. Багато так званих $\Pi\Sigma$ пруверів імплементують $\mathsf{MLTT}-75$ серед таких як: $\Pi\Sigma$[2], $\Pi\forall$[3].

Definition 42. (Мовна категорія $\mathsf{O}_{\mathsf{MLTT}-75}$).

$$\mathsf{O}_{\mathsf{MLTT}-75} = \begin{cases} \mathsf{Ob} : \{\ \mathtt{maybe}\ \mathsf{MLTT}-75\ \} \\ \mathsf{Hom} : \begin{cases} \mathtt{type}, \mathtt{norm} : \mathsf{Ob} \to \mathsf{Ob} \\ \mathtt{certify} : \mathsf{Ob} \to \mathsf{Ob} = \mathtt{type} \circ \mathtt{norm} \end{cases} \end{cases} \tag{2.8}$$

Definition 43. (Синтаксис мовної категорії $\mathsf{O}_{\mathsf{MLTT}-75}$). Мова $\mathsf{O}_{\mathsf{MLTT}-75}$ включає в себе синтаксиси трьох типів теорії Мартіна-Льофа: O_{Π}, O_{Σ}, $\mathsf{O}_{=}$.

```
def MLTT : U
 := inductive { forall (_: Pi MLTT)
              | sigma (_: Sigma MLTT)
              | id (_: Id MLTT)
              }
```

Definition 44. (Синтаксичне дерево O_{Σ}). Також можна до чистої системи додати Σ-тип, піднявши типову систему до мови $\mathsf{O}_{\mathsf{MLTT}-72}$ або $\mathsf{O}_{\Pi\Sigma}$:

```
def Sigma (lang: U) : U
 := inductive { sigma (n: name) (a b: lang)
              | pair (a b: lang)
              | fst (p: lang)
              | snd (p: lang)
              }
```

Definition 45. (Синтаксичне дерево $\mathsf{O}_{=}$). Додавши тип рівності можно підняти систему ще на одну сходинку, до $\mathsf{O}_{\mathsf{MLTT}-75}$ або $\mathsf{O}_{\Pi\Sigma=}$:

```
def Id (lang: U) : U
 := inductive { identity (t a b: lang)
              | id_intro (a b: lang)
              | id_elim (a b c d e: lang)
              | id_compute (a b c d e: lang)
              }
```

$\mathsf{O}_{=}$ не містить η-правила.

[2] https://github.com/zlizta/pisigma-0-2-2
[3] https://github.com/sweirich/pi-forall

2.3.3 Система індуктивних типів MLTT-80

MLTT-80 покладена в основу CCHM верифікатора.

Definition 46. (Мовна категорія $\mathsf{O}_{\mathsf{MLTT-80}}$).

$$\mathsf{O}_{\mathsf{MLTT-80}} = \begin{cases} \mathsf{Ob} : \{\ \mathsf{X} : \mathsf{maybe\ PM}, \mathsf{target} : \mathsf{maybe\ CPS}\ \} \\ \mathsf{Hom} : \begin{cases} \mathsf{type}, \mathsf{norm}, \mathsf{induction} : \mathsf{X} \to \mathsf{X}, \mathsf{extract} : \mathsf{X} \to \mathsf{target} \\ \mathsf{certify} : \mathsf{X} \to \mathsf{target} \\ \mathsf{cerfity} = \mathsf{type} \circ \mathsf{norm} \circ \mathsf{induction} \circ \mathsf{extract} \end{cases} \end{cases} \tag{2.9}$$

Мова індуктивних типів дозволяє безпосередньо кодувати індуктивні типи, не використовуючи схеми кодування Бома, містить усі попередні мовні синтаксиси: $\mathsf{O}_{=}$, O_{Σ}, O_{Π} та синтаксиси O_0, O_1, O_2, O_W.

Definition 47. (Синтаксичне дерево мовної категорії $\mathsf{O}_{\mathsf{MLTT-80}}$).

```
def MLTT-80 := Per
def Per : U
 := inductive { forall (_: Pi Per)
              | sigma (_: Sigma Per)
              | id (_: Id Per)
              | 0 (_: Empty Per)
              | 1 (_: Unit Per)
              | 2 (_: Bool Per)
              | W (_: W Per)
              }

def W (lang: U) : U
 := inductive { W_Form (n: name) (a b: lang)
              | W_Sup (a b: lang)
              | W_Ind (a b c: lang)
              }

def Empty (lang: U) : U := inductive { 0_Ind (a: lang) }
def Unit  (lang: U) : U := inductive { unit | star | 1_Ind (a: lang) }
def Bool  (lang: U) : U := inductive { bool | true | false | 2_Ind (a: lang) }
```

2.3.4 Система індуктивних типів CIC

Definition 48. (Мовна категорія $\mathsf{O}_{\mathsf{CIC}}$).

$$\mathsf{O}_{\mathsf{CIC}} = \begin{cases} \mathsf{Ob} : \{\ \mathsf{X} : \mathtt{maybe}\ \mathsf{PM}, \mathtt{target} : \mathtt{maybe}\ \mathsf{CPS}\ \} \\ \mathsf{Hom} : \begin{cases} \mathtt{type}, \mathtt{norm}, \mathtt{induction} : \mathsf{X} \to \mathsf{X}, \mathtt{extract} : \mathsf{X} \to \mathtt{target} \\ \mathtt{certify} : \mathsf{X} \to \mathtt{target} \\ \mathtt{cerfity} = \mathtt{type} \circ \mathtt{norm} \circ \mathtt{induction} \circ \mathtt{extract} \end{cases} \end{cases} \tag{2.10}$$

Мова індуктивних типів дозволяє безпосередньо кодувати індуктивні типи, не використовуючи схеми кодування Бома, містить усі попередні мовні синтаксиси: $\mathsf{O}_{=}$, O_{Σ}, O_{Π}.

Definition 49. (Синтаксичне дерево мовної категорії $\mathsf{O}_{\mathsf{CIC}}$).

Головним чином, система загальних індуктивних схем предбачає три основних компоненти: 1) верифікатор строго позитивних схем; 2) верифікатор завершуваності рекурсивної перевірки рекурсивних схем; 3) верифікатор взаємної рекурсії.

```
def Frank := inductive { forall (_: Pi Frank) | ind (_: Ind Frank) }
def Christine := CIC
def CIC : U
 := inductive { forall (_: Pi CIC)
              | sigma (_: Sigma CIC)
              | id (_: Id CIC)
              | prop (_: Id CIC)
              | ind (_: Ind CIC)
              }
```

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

Definition 50. (Синтаксичне дерево $\mathsf{O}_{\mathsf{IND}}$). Правило формації, конструктора та елімінатора визначається синтаксичним деревом $\mathsf{O}_{\mathsf{IND}}$:

```
def Ind (lang: U) : U
 := inductive { formation (_: Inductive lang)
              | constructor (_: list (triple Nat Inductive lang))
              | eliminator (t: Inductive) (a b: lang) (cases: list lang)
              }

def Inductive (lang: U) : U
 := Σ (name : string)
      (params : list (prod name lang))
      (level : Nat)
      (constrs : list (prod (Nat lang))), 1
```

2.3.5 Гомотопічна система типів HTS

Головним чином, гомотопічна система складається з наступних частин: 1) два всесвіти fibrant та pretype; 2) MLTT-80; 3) CCHM розширення.

Definition 51. (Мовна категорія $\mathsf{O}_{\mathsf{HTS}}$).

$$\mathsf{O}_{\mathsf{HTS}} = \begin{cases} \mathsf{Ob} : \{\ \mathit{maybe}\ \mathsf{HTS}\ \} \\ \mathsf{Hom} : \begin{cases} \mathsf{type}, \mathsf{norm} : \mathsf{Ob} \to \mathsf{Ob} \\ \mathsf{certify} : \mathsf{Ob} \to \mathsf{Ob} = \mathsf{type} \circ \mathsf{norm} \end{cases} \end{cases}$$

Definition 52. (Синтаксис мовної категорії $\mathsf{O}_{\mathsf{HTS}}$). Синтаксис гомотопічної мовної категорії містить усі попередні мовні синтаксиси: O_{I}, O_{W}, $\mathsf{O}_{=}$, O_{Σ}, O_{Π}:

```
def HTS : U
 := inductive { forall (_: Pi HTS)
              | sigma (_: Sigma HTS)
              | id (_: Id HTS)
              | prop (_: Id HTS)
              | 0 (_: Empty HTS)
              | 1 (_: Unit HTS)
              | 2 (_: Bool HTS)
              | W (_: W HTS)
              | homotopy (_: Homotopy HTS)
              }
```

Гомотопічна мова наслідує $\mathsf{O}_{\mathsf{MLTT-80}}$ 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.

Definition 53. (Синтаксичне дерево O_{I}).

```
def CCHM (lang: U) : U
 := inductive { pretype (n: nat)
              | PathP (_: lang) | PLam (_: lang) | PApp (f a: lang)
              | I | 0 | 1 | And (a b: lang) | Or (a b: lang) | Neg (_: lang)
              | Transp (a b: lang) | HComp (a b c d: lang)
              | Partial (_: lang) | PartialP (a b: lang) | System (_: lang)
              | Sub (a b c: lang) | Inc (a b: lang) | Ouc (: lang)
              | Glue (: lang) | GlueElem (a b c: lang) | Unglue (_: lang)
              }
```

Таким чином, $\mathsf{O}_{\mathsf{HTS}}$ містить два Id-типа, один унаслідований від $\mathsf{O}_{=}$ (з модифікованою обчислювальною семантикою), а інший Interval який міститься в синтаксичному дереві O_{I}.

2.3.6 Висновки

Таким чином ми здійснили спектралізацію або іншими словами розклали усі існуючі немодальні формальні системи типів у спектральну категорію.

Системи фібраційного типу Groupoïd Infinity для математичного представлення, сертифікації (доведення теорем), з екстракцією в сертифікований інтерпретатор та його середовища виконання.

Основним дослідницьким продуктом є формальне середовище виконання AXIO/1, яке здатне запускати прості лямбда-програми на верифікованому інтерпретаторі Joe. Але ніхто не може обмежити розвиток своїх власних вищих мов з екстракцією в цей інтерпретатор. Метою створення AXIO/1 є утримування процесу розробки під одним авторіті процесі управління відкритим програмним забезпеченням (Кафедральна модель управління).

AXIO/1 складається з середовища виконання та його мов Joe, Alice, Bob для розробки системного програмного забезпечення, які використовують стандартний синтаксис ML; і більш високі мови та їх екстрактори для виконання Alonzo, Henk, Per, Anders для доведення теорем, які використовують Lean-подібний синтаксис. Неверифікований CPS інтерпретатор реалізований на мові Rust.

Інтерпретатори і системні мови

Верифікований лямбда-інтерпретатор і паралельне обчислення матриць. Мови Joe, Bob і Alice мають уніфікований синтаксис Standard ML.

Joe

Джо є сертифікованим інтерпретатором байт-коду стекової віртуальної машини та компілятором коду Intel/ARM/SM90.

[1] — MinCaml, [2] — CoqASM. [3] — Verified LISP Interpreter, [4] — Kind, [5] — O-CPS/Rust.

```
fun a (0, n) = n + 1
  | a (m, 0) = a (m - 1, 1)
  | a (m, n) = a (m - 1, a (m, n - 1))
```

Bob

Bob — це паралельне неблокуюче середовище виконання з нульовим копіюванням та курсорами CAS-кусрорами [4,5] з характеристиками для вбудовуваних систем реального часу.

[5] — Kernel, [6] — Pony, [7] — Erlang.

```
fun proc =
let val p0 = pub(0,8)
    val s1 = sub(0,p0)
    val s2 = sub(0,p0)
 in send(p0,11);
    send(p0,12);
    [ receive(s1);
      receive(s2);
      receive(s1);
      receive(s2)
    ]
    end
```

Alice

Alice — це числення лінійних типів із частковими дробами [6] для програмування рівня 3 BLAS.

```
fun simpleConvolution (i n: int) (x0: float) (write w: vector float)
  : vector float
  = begin
      if n = i then result.emit(write),
      a = [w0,w1,w2] = w.get(0,3),
      b = [x0,x1,x2] = [ x0 | write.get(i,2) ],
      write.set(i, Dotp(a,b)),
      simpleConvolution((i + 1),n,x1,write,w)
    end
```

Чисті фібраційні мови програмування (вищих порядків)

Чисті фібраційні мови мають уніфікований синтаксис Lean-подібної мови.

Alonso

Alonzo — це система типу STLC-40 як приклад основного числення, відкритого до фібраційних ΠΣ пруверів.

[1] STLC-40 — Проста теорія типів

```
def zero : (T → T) → T → T := λ (s: T → T) (z: T), z
def succ : ((T → T) → T → T) → ((T → T) → T → T)
 := λ (w: (T → T) → T → T) (y: T → T) (x: T), y (w y x)
```

Henk

Henk — це система чистого типу (PTS-91) у стилі числення індуктивних конструкцій Кокванда/Х'юета (CoC-88) з нескінченною кількістю всесвітів. Henk також підтримує синтаксис AUTOMATH (AUT-68).

[1] AUT-68 — AUTOMATH 1968, [2] CoC-88 — числення конструкцій, [3] PTS-91 — система чистого типу (Π).

```
def N := Π (A : U), (A → A) → A → A
def zero : N := λ (A : U) (S : A → A) (Z : A), Z
def succ : N -> N := λ (n : N) (A : U) (S : A → A) (Z : A), S (n A S Z)
def plus (m n : N) : N := λ (A : U) (S : A → A) (Z : A), m A S (n A S Z)
def mult (m n : N) : N := λ (A : U) (S : A → A) (Z : A), m A (n A S) Z
def pow (m n : N) : N := λ (A : U) (S : A → A) (Z : A), n (A →
A) (m A) S Z
```

Per

Пер є ΠΣ (MLTT-72) прувером із численням індуктивних конструкцій та типами тотожності (MLTT-75). Природне розширення CoC до CIC було зроблено Франком Пфеннінгом і Крістін Полін (IND-89).

[1] Mini-TT — реалізація OCaml, [2] MLTT-72 — Πi, Sigma, [3] MLTT-75 — Pi, Sigma, Id, [4] MLTT-80 — 0, 1, 2, W, Pi, Sigma, Id, [5] PP-89 — Індуктивно визначені типи, [6] CIC-2015 — Числення індуктивних конструкцій.

```
def empty       : U := inductive { }
def L¹ (A : U) : U := inductive { nil | cons (head: A) (tail: L¹ A) }
def S¹          : U := inductive { base | loop : Equ S¹ base base }

def quot (A: U) (R : A -> A -> U) : U
 := inductive { quotient (a: A)
              | identification (a b: A) (r: R a b)
              : Equ (quot A R) (quotient a) (quotient b)
              }
```

Anders

Anders — це система гомотопічних типів (HTS-2013) із примітивами суворої рівності та кубічної Agda (CCHM-2016).

[1] HTS-2013 — система гомотопічних типів, [2] BCH-2014 — Кубічні набори, [3] CCHM-2015 — Система кубічного типу, [4] ОП-2016 — Топосні аксіоми, [5] CHM-2017 — Рівняння Губера, [6] BMA-2017 — Кубічна Агда.

```
def idfun  (A : U) : A → A := (λ (a : A), a
def idfun' (A : U) : A → A := transp (<i> A) 0
def idfun'' (A : U) : A → A := (λ (a : A), hcomp A 0 (λ (i : I), []) a
def isFiberBundle (B: U) (p: B → U) (F: U): U
 := Σ (v: U) (w: surjective v B), (Π (x: v), PathP (<_>U) (p (w.1 x)) F)
def ~~ (X : U) (a x' : X) : U := Path (ℑ X) (ι X a) (ι X x')
def 𝔻 (X : U) (a : X) : U := Σ(x' : X), ~~ X a x'
def unitDisc (X : U) (x : ℑ X) : U := Σ(x' : X), Path (ℑ X) x (ι X x')
def starDisc (X : U) (x : X) : 𝔻 X x := (x, idp (ℑ X) (ι X x))
def T∞ (A : U) : U := Σ(a : A), 𝔻 A a
def inf−prox−ap (X Y : U) (f : X → Y) (x x' : X) (p : ~~ X x x')
  : ~~ Y (f x) (f x') := <i> ℑ−app X Y f (p @ i)
def d (X Y : U) (f : X → Y) (x : X) (ϵ : 𝔻 X x) : 𝔻 Y (f x)
 := (f ϵ.1, inf−prox−ap X Y f x ϵ.1 ϵ.2)
def T∞−map (X Y : U) (f : X → Y) (τ : T∞ X) : T∞ Y
 := (f τ.1, d X Y f τ.1 τ.2)
def is−homogeneous (A : U)
 := Σ (e : A) (t : A → equiv A A),
    Π (x : A), Path A ((t x).1 e) x
```

Urs

Urs — це еквіваріантна система супергомотопічних типів з ферміонними та бозонними модальностями, вбудованими в верифікатор.

[1] R-HoTT — Rezk Infinity Categories, [2] G-HoTT — Guarded Cubical, [3] L-HoTT — Linear HoTT, [4] ES-HoTT — Equivariant Super HoTT.

2.4 Когомології мов програмування

Ця робота презентує і формалізує когомологічний підхід до синтаксисів мов програмування, представлених як вищі індуктивні типи (HIT), інтерпретовані як CW-комплекси. Правила β та η моделюються як рівності (1-клітини та 2-клітини) у цих структурах, що дозволяє будувати ланцюгові комплекси та когомологічні групи. Ці групи відображають гомотопічні інваріанти синтаксисів, такі як конструктори, редукції та вищі гомотопії, у межах спектральної моноїдальної категорії мов програмування. Підхід розширюється до спектральної послідовності мов, від простих чистих систем типів до складних гомотопічних систем типів.

Ця робота розвиває когомологічну теорію для гомотопічних синтаксисів, визначаючи ланцюгові комплекси та когомологічні групи для відображення їх структурних і гомотопічних властивостей. Підхід реалізується в межах *спектральної моноїдальної категорії* мов програмування, де мови є

об'єктами, а трансформації (наприклад, компіляція, верифікація) — морфізмами.

2.4.1 Синтаксичні CW-комплекси

Definition 54 (Синтаксичний CW-комплекс)**.** Синтаксичне дерево O_x для мови програмування (наприклад, O_Π, O_Σ, O_I) є вищим індуктивним типом, що складається з:

- *0-клітини*: Конструктори синтаксису (наприклад, variable, lambda, application).
- *1-клітини*: Правила β, що визначають рівності між термами (наприклад, $(\lambda x.M)N = M[N/x]$).
- *2-клітини*: Правила η, що визначають гомотопії між рівностями (наприклад, $\lambda x.fx = f$).
- *Вищі клітини*: Вищі гомотопії, що виникають у гомотопічних системах типів (наприклад, Path-типи в O_{HTS}).

Example 1 (CW-комплекс для O_Π)**.** Синтаксичне дерево для Π-типу, O_Π, визначається як:

$$\begin{aligned} O_\Pi ::= {} & \text{fibrant}(n : \mathbb{N}) \mid \text{variable}(x : \text{name}, l : \mathbb{N}) \mid \text{pi}(x : \text{name}, l : \mathbb{N}, f : O_\Pi) \\ & \mid \text{lambda}(x : \text{name}, l : \mathbb{N}, f : O_\Pi) \mid \text{application}(f, a : O_\Pi) \\ & \mid \beta(x : \text{name}, l : \mathbb{N}, f, a : O_\Pi) : \text{application}(\text{lambda}(x, l, f), a) = f[a/x] \\ & \mid \eta(x : \text{name}, l : \mathbb{N}, f : O_\Pi) : \text{lambda}(x, l, \text{application}(f, \text{variable}(x, l))) = f \end{aligned}$$

Тут правила β утворюють 1-клітини, а правила η — 2-клітини, формуючи O_Π як CW-комплекс.

2.4.2 Ланцюговий комплекс синтаксисів

Definition 55 (Ланцюговий комплекс синтаксису)**.** Для синтаксичного дерева O_x ланцюговий комплекс $C_\bullet(O_x)$ визначається як:

- $C_n(O_x)$: Множина n-клітин CW-комплексу (наприклад, $C_0 = \{\text{конструктори}\}$, $C_1 = \{\beta\text{-правила}\}$, $C_2 = \{\eta\text{-правила}\}$).
- Диференціали $\partial_n : C_n \to C_{n-1}$, що відображають n-клітини на їх межі (наприклад, $\partial_1(\beta) = \text{application} - \text{substitution}$).
- Умова: $\partial_{n-1} \circ \partial_n = 0$, що забезпечує композиційність трансформацій.

Example 2 (Ланцюговий комплекс для O_Π)**.** Для O_Π:

- $C_0(O_\Pi) = \{\text{variable}, \text{pi}, \text{lambda}, \text{application}\}$.
- $C_1(O_\Pi) = \{\beta : (\lambda x.M)N \to M[N/x]\}$.

- $C_2(O_\Pi) = \{\eta : \lambda x.fx \to f\}$.
- Диференціал: $\partial_1(\beta) = \text{application}(\text{lambda}(x, l, f), a) - f[a/x]$.
- Диференціал: $\partial_2(\eta) = \text{lambda}(x, l, \text{application}(f, \text{variable}(x, l))) - f$.

2.4.3 Когомологічні групи

Definition 56 (Когомологічні групи)**.** Когомологічні групи синтаксичного дерева O_x визначаються як:

$$H_n(O_x) = \ker(\partial_n)/\text{im}(\partial_{n+1})$$

де:

- $H_0(O_x)$: Зв'язні компоненти, що представляють незалежні конструктори.
- $H_1(O_x)$: Цикли, що представляють нетривіальні β-правила.
- $H_2(O_x)$: Гомотопічні класи η-правил, що відображають вищі зв'язки.

Example 3 (Когомології O_{HTS})**.** Для гомотопічної системи типів O_{HTS}, що включає Path-типи та операції композиції:

- $H_0(O_{HTS})$: Конструктори (Π, Σ, $=$, W, I).
- $H_1(O_{HTS})$: Рівності, згенеровані β-правилами та Path-типами.
- $H_2(O_{HTS})$: Гомотопії, згенеровані η-правилами та операціями композиції (наприклад, HComp).

2.4.4 Спектральна моноїдальна категорія та когомології

Спектральна моноїдальна категорія $\mathcal{L}$ має об'єкти у вигляді мов програмування O_x та морфізми у вигляді фільтрів ($\mathcal{F}_{\text{enrich}}$, $\mathcal{F}_{\text{simp}}$ тощо). Визначаємо ланцюговий комплекс для $\mathcal{L}$:

Definition 57 (Ланцюговий комплекс $\mathcal{L}$)**.** Ланцюговий комплекс $C_\bullet(\mathcal{L})$ визначається як:

- $C_0(\mathcal{L}) = \{O_x\}$: Множина мов.
- $C_1(\mathcal{L}) = \{\mathcal{F}_{\text{enrich}}, \mathcal{F}_{\text{simp}}, \mathcal{F}_{\text{trans}}, \dots\}$: Множина фільтрів.
- $C_2(\mathcal{L})$: Природні перетворення між фільтрами (наприклад, гомотопії між компіляторами).
- Диференціали: $\partial_1(\mathcal{F}_{\text{enrich}}) = O_y - O_x$, для $\mathcal{F}_{\text{enrich}} : O_x \to O_y$.

Definition 58 (Когомології $\mathcal{L}$)**.** Когомологічні групи $H_n(\mathcal{L})$ визначаються як:

$$H_n(\mathcal{L}) = \ker(\partial_n)/\text{im}(\partial_{n+1})$$

- $H_0(\mathcal{L})$: Класи еквівалентності мов за виразністю.
- $H_1(\mathcal{L})$: Нетривіальні цикли фільтрів (наприклад, цикли нормалізації).
- $H_2(\mathcal{L})$: Гомотопічні класи природних перетворень.

2.4.5 Спектральна послідовність мов

Спектральна послідовність мов визначається як функтор:

$$O_\infty : \mathbb{N} \to \mathcal{L}, \quad O_\infty(n) = \{O_{CPS}, O_{PTS}, O_{MLTT-80}, O_{HTS}, \dots\}$$

Когомологічні групи $H_n(O_x)$ для кожної мови індукують послідовність гомоморфізмів:

$$H_n(O_{CPS}) \to H_n(O_{PTS}) \to H_n(O_{MLTT-80}) \to H_n(O_{HTS})$$

Ця послідовність відображає гомотопічну еволюцію синтаксисів мов.

2.4.6 Висновки

Цей підхід пропонує нову теорію аналізу синтаксисів мов програмування за допомогою когомологій, де правила β та η моделюються як рівності у вищих індуктивних типах. Інтерпретація синтаксисів як CW-комплексів дозволяє визначити ланцюгові комплекси та когомологічні групи, що відображають гомотопічні інваріанти. Інтеграція в спектральну моноїдальну категорію дає змогу вивчати трансформації мов та їх ієрархічну структуру.

Розділ 3

Система мов середовища виконання

Присвячується автору Erlang

Джо Армстронгу

Третій розділ описує розвиток концептуальної моделі системи доведення теорем як сукупності формальних середовищ виконання, кожне наступе з яких, складніше за попереднє, має свою операційну семантику, та наслідує усі властивості попередніх операційних середовищ послідовності.

Вступне слово

3.1 Інтерпретатор як основна лямбда-система

Мінімальна мова системи $\mathsf{O}_{\mathsf{CPS}_\lambda}$ визначається простим синтаксичним деревом:

```
def CPSλ : U
 := inductive { 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.

Табл. 3.1: Заміри на інтерпретаторах ландшафту атаки

Мова	**Fac(5) в нс**
Rust	0
Java	3
PyPy	8
CPS	291
Python	537
K	756/635
Erlang	10699/1806/436/9
LuaJIT	33856

Табл. 3.2: Заміри на інтерпретаторах ландшафту атаки

Мова	**Akk(3,4) в мкс**
CPS	635
Rust	8,968

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 есперименталний інтерпретатор без байт-коду, але з вирівняним по словам архітерктури стріму команд, які є безпосередньою машинною презентацією конструкторів індуктивних типів (enum) мови Rust. Наступні результати були отримані після неотпимізованої версії інтерпретатора при обчисленні факторіала (5) та функції Акермана у точці (3,4).

Ключовим викликом тут стали лінійні типи мови Rust, які не дозволяють звертатися до ссилок, які вже були оброблені, а це впливає на всю архітектуру тензорного преставлення змінних в мові інтерпретатор $\mathrm{O}_{\mathrm{CPS}}$, яка наслідує певним чином мову K.

3.1.1 Векторизація засобами мови Rust

```
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
```

3.1.2 Байт-код інтерпретатора

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

```
def Lazy : U
 := inductive { 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.

```
def Cont : U
 := inductive { Expressions (a: AST) (v: Option (Iter AST)) (c: 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) (c: Cont)
              | Call (a: AST) (i: Nat) (cont: Cont)
              | Return
              | Intercore (m: Message) (cont: Cont)
              | Yield (cont: Cont)
              }
```

3.1.3 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 (куди 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).

```
def AST : U
 := inductive { Atom (a: Scalar)
              | Vector (a: Vec AST)
              }

def Value : U
 := inductive { Nil
              | SymbolInt (a: u16)
              | SequenceInt (a: u16)
              | Number (a: i64)
              | Float (a: f64)
              | VecNumber (Vec i64)
              | VecFloat (Vec f64)
              }
```

```
def Scalar : U
 := inductive { 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)
              }
```

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.

3.2 Система числення процесів SMP async

3.2.1 Операційна система

Перелічимо основні властивості операційної системи (прототип якої опублікований на Github[1]).

3.2.2 Властивості

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

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.

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

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 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]];
```

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.

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

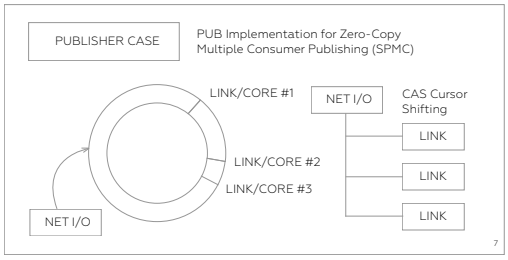


Рис. 3.1: Кільцева статична черга з CAS-курсором для публікації

(нерухомості даних) та відсутності копіювання в подальшій роботі, дані залишаються в черзі, а рухаються та передаються лише курсори на типизовані послідовності даних.

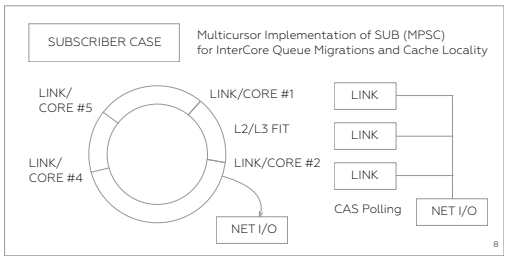


Рис. 3.2: Кільцева статична черга з CAS-курсором для згортки

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 виглядає як публікація у

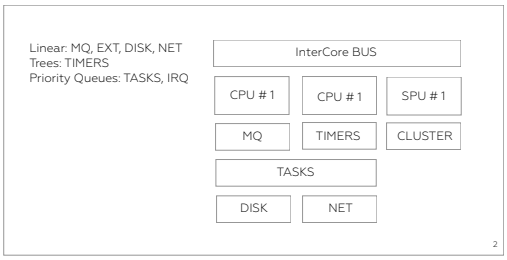


Рис. 3.3: Система процесорних ядер та реакторів

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 нескінченні (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 самобалансуючі дерева, BTree дерева тощо).

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.

fun pub(capacity: int): int

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

fun sub(publisher: int): int

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

fun spawn(core: int , program: code, cursors: array int): int

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.

fun kill (process: int): int

Денонсація процесора на реаторі.

fun send(writer: int , data: binary): int

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

fun receive(reader: int)

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

3.2.3 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 $\mathsf{O}_{\mathsf{CPS}}$ програми.

Черга для публікації

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

Черга для читання

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

Існує дві спецільні задачі: InterCore задача, написана на Rust, яка запускається на всіх ядрах при запуску системи, а також CPS-інтерпретор головного термінала системи, який 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 низькорівневих 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.

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>>,
}
```

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>>>,
}
```

Планувальник

Планувальник репрезентує ядра процесара, 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,
}
```

3.2.4 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,
}
```

3.3 Система числення тензорів AVX

Для реалізації мови програмування високого рівня на BLAS Level 3 бекендом була вибрана мова NumLin, серед інших: 1) Ling, 2) Guarded Cubical, 3) A Fibrational Framework for Substructural and Modal Logics, 4) APL-like interpreter in Rust (дана робота), 5) Futhark.

```
def AVX-512 : U
 := inductive { Star | True | False
              | Variable (_: Var)
              | Prim (_: Builtin)
              | Int (_: nat) | Float (_: float)
              | Lambda (a: Var) (b: Linear) (c: Exp)
              | App (a b: Exp)
              | Pair (a b: Var) (c d: Exp)
              | Consume (a: Var) (b c: Exp)
              | Gen (a: Var) (b: Exp)
              | Spec (a: Exp) (b: Fraction)
              | Fix (a b: Var) (c d: Linear) (e: Exp)
              | If (a b c: Exp)
              | Let (a: Var) (b c: Exp)
              }
```

3.4 Висновки

Перша стадія реалізації класичного лінивого інтерпретатора з CPS семантикою була виконана як MVP трейдингової HFT платформи. Наступна стадія — виконання верифікованого інтерпретатора (віртуальної машини) та компілятора (в неї) Standard ML мови на основі компілятора Joe (MinCaml).

Розділ 4

Бібліотека середовища виконання

Присвячується автору
формальної системи F

Жану-Іву Жирару

Після побудови в розділі 3 формального середовища виконання, яке складається з операційної системи у якій виконуються CPS-інтерпретатори з формальною системою вводу-виводу IO, можна зразу переходити до базової бібліотеки середовища виконання.

Даний розділ формалізує інтерфейс прикладного програмування та систему бібліотек часу виконання для забезпечення потреб побудови гетерогенних систем та сервісів.

Вступне слово

Так чи інакше для дослідження будь-якої базової бібліотеки середовища виконання доведеться зустрітися з теорією яка стоїть за System F. Навіть базова бібліотека фундаментальної вищої мови PTS в сутності потребує для свого кодування лише системи Ф, так як є безпосереднім портом з мови Haskell. Тому доречно використовувати у якості проміжної типової системи систему Ф Жирара як цільову систему для екстрактів з вищих мов, таких як HTS (якщо такі екстракти існують для окремих програм).

4.1 Загальні принципи

N2O.DEV — це формальна філософія та інженерна вправа водночас. Вона обмежує автора бути ефективним та точним не втрачаючи при цьому повноти та функціональності. Це накшталт внутрішньої дисципліні

при проектуванні програмного забезпечення. Ця філософія багато років застосовується на практиці для побудови систем SYNRC, та визначає стандартний мінімальний набір для демонстрації однієї з сучасних моделей реактивного веб-програмування, яка включає: веб-сокет веб-фреймвок з бінарною серілізацією, пушами та контролем DOM елементів зі сторони сервера. N2O.DEV вчить будувати прості та надійні системи на будь-якій мові програмування.

4.2 Формальна специфікація

Формальне середовище виконання визначає структуру операційних середовищ (runtimes) як операційну систему лямбда-інтерпретаторів які працюють на паралельному обчислювальному середовищі (ядрах процесорів). Кожне з ядер процесорів виконує в нескінченному циклі команди лямбда-інтерпретаторів, переключаючи через певний проміжок часу на потік команд іншого інтерпретатора. Таке визначення дає змогу вбудувати цю структуру у віртуальну машину Erlang: 1) Головний процес додатку; 2) Супервізор додатку; 3) Проміжні супервізори; 4) Кінцеві пул процесорів повідомлень.

Тут визначена специфікація програмного забезпечення усіх рівнів прикладної моделі для підприємств на функціональних мовах програмування. Ця специфікація визначає правила побудови WebSocket сервера, бінарного серіалізатора та веб-фреймворку визначеному формальними протоколами. Промислові версії також підтримують систему управління бізнес-процесами та ефективне сховище з глобальним простором ключів.

Серверні та клієнтські мови

Мови програмування розділені на чотири рівня як для клієнтської (мобільної та веб розробки) так і для серверної розробки (бекенд системи). Нульовий рівень — тотальні формальні алгебраїчні мови програмування, що забезбечують повноту, функціональність та доведення властивостей прогрм згідно сучасних уявлень про математичне моделювання та системи залежних типів побудованих на розшаруваннях. Перший рівень — формальні функціональні мови програмування, як правило System F, System F_ω які успішно використовуються в промисловості та забезпечують достатньо формальний запис який піддається масштабуванню у великих командах завдяки потужному ядру компіляторів. Другий рівень — неформальні (без формальної операційної чи денотаційної семантики) чи формальних верифікаторів, які проте успішно використовуються в промисловості, можуть бути як з розвиненими системами типів з узагальненими шаблонами та типами сумами, так і однотипними мовами програмування з динамічною типізацією. Третій рівень — мови які погано піддаються масштабуванню у промисловому виробництві (на основі спостережень за власним досвідом).

Табл. 4.1: Перелік кваліфікаційних рівніх верифікації

Сторона/Рівень	**Приклади мов**
клієнт/3	JavaScript, Lua
клієнт/2	Swift, Kotlin, TypeScript
клієнт/1	UrWeb, OCaml, PureScript
клієнт/0	Kind, PTS
сервер/3	PHP, Python, Perl, Ruby
сервер/2	Erlang, Elixir
сервер/1	SML, Scala, Haskell, F#, Rust, Hamler
сервер/0	Coq, Agda, Lean, MLTT/HTS

Обрані мови реалізації

Тут перелічені мови, на яких реалізовано та апробовано N2O.DEV.

Standard ML[1]. В академічних цілях Марат Хафізов створив за специфікаєю N2O/NITRO порт на мови Standard ML (SML/NJ та MLton). Ця робота представлена Github організацією O1 в структурі N2O.

Haskell[2]. Перший експеримент з формалілазції N2O в систему F була робота Андрія Мельникова. Пізніше, більш повну версію з NITRO протокол запропонував Марат Хафізов, ця версія представлена на Github як організація O3.

F#[3]. Також у якості вправи Siegmentation Fault зробив порт NITRO на мову F# разом з ETF кодуванням. Ці напрацювання представлені в Github організації O61.

Lean[4]. У якості більшо формальної платформи з залежними типами, мова Lean 4 від Леонардо де Мура з Microsoft. Siegmentation Fault автор порта, який представлений Github організацією O89 та сайтом lean4.dev.

Erlang[5]. Основна промислова платформа, яка в повній мірі реалізує специфікацію N2O.DEV.

Elixir[6]. Адаптація N2O.DEV для мови Elixir. Основна імплементація не змінена, постійно підтримується публікація пакетів на HEX.PM.

Hamler. Нова формальна платоформа на базі PureScript для віртуальної машини Erlang. В подальшому цей розділ буде присвячений імплементації та специфікації на мові Hamler (варіація PureScript з бекендом в Erlang Core).

Авторськими вважаються імплементації на мовах Erlang (Elixir) та Hamler (PureScript). Інші представлені імплементації вважаються сертифікаваними формальними референсними моделями.

[1] `https://github.com/o1`
[2] `https://github.com/o3`
[3] `https://ws.erp.uno`
[4] `https://github.com/o89`
[5] `https://github.com/synrc`
[6] `https://github.com/synrc`

Табл. 4.2: Перелік мов для яких існує версія базової бібліотеки

Мова/Платформа	**Набір реалізованих компонент**
Erlang/OTP	N2O, BPE, KVS, NITRO, MAD, FORM
Elixir/OTP	N2O, BPE, KVS, NITRO, FORM
Hamler/OTP	RT, BASE, N2O, BPE, KVS, NITRO
Standard ML	N2O, NITRO, ETF
Haskell	N2O, NITRO, ETF
F#	N2O, NITRO, ETF
Lean 4	N2O, NITRO, MAD, ETF

Протягом 8 років автор практикувався аби адаптувати бібліотеку середовища виконання до основних формальних або напівформальних мов System F, які принаймі написані на мовах які теж є мовами System F (SML, F#, Haskell). Сторонніми дослідниками була навіть зроблена адаптація для Lean 4[7]. В процесі цьої багаторічної вправи стало зрозумілим, що досягти на виробництві тієї якості, яка досягається в середовищі виконання Erlang/OTP майже неможливо. Тому модель базової бібліотеки, спочатку адаптувалася з оригінальної мови Erlang (2013) для мови Elixir (2018), і потім для мови Hamler (2021), що є варіацією PureScript (System F_ω). Всього існує 7 моделей для 7 мов програмування базової бібліотеки представленої в цій роботі.

4.3 Пакети формального середовища

Тут представлена модель бібліотеки формального середовища виконання, яке скаладається з JIT-інтерпретатора, потужної SMP-системе акторів з неблокуючими курсорами черг повідомлень для системи процесів 1:1 (процес-черга, кожен процес має свою чергу). Така спрощена модель дещо відрізняється від моделі CSP, CCS та класичного числення процесів (Pi Calculus) проте протягом цих 8 років на практиці стало зрозумілим, що модель Erlang/OTP більш гнучка до масштабування та стійка до помилок.

Іншими словами модель в System F базової бібліотеки середовища виконання формально визначає цей прошарок уніфікованого мовного середовища. А модель процесів хоча формально не відображає семантику коіндукції (стерта інформація), проте досі синтаксис мовного середовища представленого в розділі 2 є сумісним з моделлю Erlang/OTP яка є основною платформою, що підтримується у виробництві.

[7] `https://o89.github.io/lean4.dev/`

4.3.1 Структури даних BASE

Структури даних представлені додатком BASE. Основні модулі додатку: List, Array, Atomics, Binary, Bool, Char, Counters, DateTime, Digraph, Enum, Eq, Float, Int, Map, Maybe, Ord, OrdDict, OrdSet, Pid, Queue, Read, Record, Regex, Set, Time, Tuple.

Сигнатури додатка BASE максимально сумісні з базовою бібліотекою Hamler.

4.3.2 Сервіси середовища виконання RT

Сервіси середовища виконання представлені додатком RT, іменні простори якого дещо відрізняються від відповідних сигнатур базової бібліотеки мови Hamler.

Database

Модулі простору Database: Mnesia, ETS.

```
axiom all : IO (List TableId)
axiom browse : IO String
axiom delete : Π (k: U), TableId -> k -> IO Unit
axiom first : Π (t: TableId) (k: U), IO (Maybe k)
axiom last : Π (k: U), TableId -> IO (Maybe k)
axiom foldl : Π (v acc: U), (v -> acc -> acc) -> acc -> TableId -> IO acc
axiom foldr : Π (v acc: U), (v -> acc -> acc) -> acc -> TableId -> IO acc
axiom insert : Π (v: U), TableId -> v -> IO Boolean
axiom lookup : Π (k v: U), TableId -> k -> IO (List v)
axiom member : Π (k: U), TableId -> k -> IO Boolean
axiom new : Atom -> TableOptions -> IO TableId
axiom next : Π (k: U), TableId -> k -> IO (Maybe k)
axiom prev : Π (k: U), TableId -> k -> IO (Maybe k)
axiom rename : TableId -> Atom -> IO Atom
axiom take : Π (k v: U), TableId -> k -> IO (List v)
axiom match : Π (a v: U), TableId -> a -> IO (List v)
axiom slot : Π (v: U), TableId -> Integer -> IO (Maybe (List v))
```

Filesystem

Модулі простору Filesystem: Dir, File, FilePath, IO, Active.

OS

Модулі простору OS: Env, Init, Shell.

Process

Модулі простору Process: Application, Supervisor, Dict, GenServer, Spawn, Process, Timer.

Network

Модулі простору Network: TCP, UDP, TLS, Inet, WebSocket.

4.3.3 Прикладні протоколи N2O

Сервіси веб-сокет сервера, представлені додатком N2O: N2O, PI, Proto, Ring, MQTT, WS, TCP, Heart, Syn, FTP, NITRO, ETF, GCM, Session, Cache.

Модуль N2O пропонує ряд сервісів зручних та вивірених в промисловій роботі для побудови сервісних протоколів що вбудовуються в цикли сучасних TCP, QUIC, UDP, MQTT, WebSocket серверів, та в процесі своєї роботи можуть стартувати додаткові процеси для обробки інформації під супервізією середовища виконання.

```
axiom pickle : Binary -> Binary
axiom depickle : Binary -> Binary
axiom encode : Π (k: U), k -> Binary
axiom decode : Π (k: U), Binary -> IO k
axiom reg: Π (k: U), k -> IO k
axiom unreg : Π (k: U), k -> IO k
axiom send : Π (k v z: U), k -> v -> IO z
axiom getSession : Π (k v: U), k -> IO v
axiom putSession : Π (k v: U), k -> v -> IO v
axiom getCache : Π (k v: U), Atom -> k -> IO v
axiom putCache : Π (k v: U), Atom -> k -> v -> IO v
```

Тут залишено лише саме необхідне, але не настільки тривіальне аби бути іграшковим. Всі функції всії сервісів можуть підмінятися в ході виконання. В сутності тут зібрані сервіси: 1) бінарної серіалізації Erlang Term Format (ETF), функції encode/decode; 2) функції симетричного шифрування AES-GCM/128 pickle/depickle; 3) функції Pub/Sub внутрішнього Erlang реєстра (syn/gproc/global, тощо); 4) функції роботи з персональними сесіями, які захищені зашифрованими токенами; 5) функції роботи з глобльним кеш-сервісом для бізнес-об'єктів.

Модуль N2O.PI абстрагує користувача від надмірного фольклору Supervisor та пропонує простіший протокол запуску асинхронних процесів.

```
data PI  = PI String Atom Atom Atom Integer RestartType
data Sup = Ok Pid String | Error String

axiom start : PI -> IO Sup
```

4.3.4 Сховище даних KVS

Сервіси сховища даних представлені додатком KVS: KVS, Stream, ST, Rocks, Mnesia, FS.

```
axiom get : Π (f k v: U), f -> k -> IO (Maybe v)
axiom put : Π (r: U), r -> IO StoreResult
axiom delete : Π (f k: U), f -> k -> StoreResult
axiom index : Π (f p v r: U), f -> Atom -> v -> List r


data Reader = Reader Integer Binary ETF String Integer
data Writer = Writer Integer Binary ETF String Integer
data StoreResult = Ok Integer String Binary
                 | Error Integer String Binary

axiom next : Reader -> IO Reader
axiom prev : Reader -> IO Reader
axiom take : Reader -> IO Reader
axiom drop : Reader -> IO Reader
axiom save : Reader -> IO Reader
axiom append : Π (f r: U), f -> r -> IO StoreResult
axiom remove : Π (f r: U), f -> r -> IO StoreResult
```

4.3.5 Бізнес-процеси BPE

Сервіси системи управління бізнес-процесами представлені додатком BPE: BPE, Event, Action, Process, Hist, Flow.

```
axiom start : Proc -> IO Sup
axiom stop : String -> IO Sup
axiom next : ProcId -> IO ProcRes
axiom load : ProcId -> IO ProcRes
axiom proc : ProcId -> IO ProcRes
axiom assign : ProcId -> IO ProcRes
axiom persist : ProcId -> Proc -> IO ProcRes
axiom amend : Π (k: U), ProcId -> k -> IO ProcRes
axiom discard : Π (k: U), ProcId -> k -> IO ProcRes
axiom modify : Π (k: U), ProcId -> k -> Atom -> IO ProcRes
axiom event : ProcId -> String -> IO ProcRes
axiom head : ProcId -> IO Hist
axiom hist : ProcId -> IO (List Hist)
axiom step : ProcId -> Atom -> IO (List Hist)
axiom docs : ProcId -> IO (List Tuple)
axiom events : ProcId -> IO (List Tuple)
axiom tasks : ProcId -> IO (List Tuple)
axiom doc : Tuple -> ProcId -> IO (List Tuple)

data ProcId = String
data Proc = Proc ProcId String
data ProcRes = Ok Integer String Binary
             | Error Integer String Binary
```

4.3.6 Контрольні елементи протоколу NITRO

Сервіси веб-фреймворка, представлені додатком NITRO: NITRO, Combo, Edit, Form, Input, Table, Actions, Render.

```
axiom q : Π (k: U), Atom -> k
axiom qc : Π (k: U), Atom -> k
axiom jse : Maybe Binary -> Binary
axiom hte : Maybe Binary -> Binary
axiom wire (actions: List Action) : IO (List Action)
axiom render (content: Either Action Element) : Binary
axiom insert_top (dom: Atom) (content: List Element) : IO (List Action)
axiom insert_bottom (dom: Atom) (content: List Element) : IO (List Action)
axiom update (dom: Atom) (content: List Element) : IO (List Action)
axiom clear (dom: Atom) : IO Unit
axiom remove (dom: Atom) : IO Unit
```

NITRO – це Nitrogen-подібний веб фреймворк для Erlang/OTP. Він може бути використаний не лише у веб-додатках, а ще і в консольних програмах, у яких потрібно зробити рендеринг HTML5.

Nitrogen Elements – це трансформатор шаблонів HTML для мови Erlang, у якому всі HTML теги рендеряться з Erlang рекордів.

Працюючи з N2O вам взагалі не потрібно користуватись HTML. Натомість ви визначаєте вашу сторінку у вигляді Erlang рекордів, відповідно сторінка генерується з перевіркою типів, під час компіляції. Це класичний підхід CGI для компільованих сторінок, який надає всі переваги перевірки помилок під час компіляції, та визначає DSL для клієнт- та серверного рендерингу.

Nitrogen Elements, за своєю природою, є примітивними UI елементами управління, які можуть бути використані для побудови Nitrogen сторінок з внутрішнім DSL Erlang-a. Вони компілюються в HTML та JavaScript. Поведінка всіх елементів контролюється на стороні сервера, а весь зв'язок між веб-переглядачем та сервером здійснюється за допомогою WebSocket каналів. Отже, вам не потрібно використовувати POST запити чи HTML форми. Тобто:

```
#textbox { id=userName, body= <<"Anonymous">> },
#panel { id=chatHistory, class=chat_history }
```

згенерує наступний html:

```
<input value="Anonymous" id="userName" type="text"/>
<div id="chatHistory" class="chat_history"></div>
```

А

```
nitro:update(loginButton,
    #button{id = loginButton,
        body = "Login",
        postback = login,
        source = [user, pass]});
```

згенетурє наступне:

```
qi('loginButton').outerHTML='<button id=\"loginButton\"
  type=\"button\">Login</button>'; { var x=qi('loginButton');
  x && x.addEventListener('click',function (event){
    event.preventDefault(); { if (validateSources(['user','pass'])) {
    ws.send(enc(tuple(atom('pickle'),bin('loginButton'),
      bin('b840bca20b3295619d1157105e355880f850bf0223f2f081549dc
      8934ecbcd3653f617bd96cc9b36b2e7a19d2d47fb8f9fbe32d3c768866
      cb9d6d85700416edf47b9b90742b0632c750a4240a62dfc56789e0f5d8
      590f9afdfb31f35fbab1563ec54fcb17a8e3bad463218d6402f1304'),
      [tuple(tuple(string('loginButton'),bin('detail')),[]),
      tuple(atom('user'),querySource('user')),
      tuple(atom('pass'),querySource('pass'))])));}
  else console.log('Validation Error');}});};
```

Для подальної інформації дивіться актуальную документацію[8]

4.4 Висновки

Кожна мова, на якій реалізовано N2O.DEV, має вбудовувати свою філософію максимально природно та компактно. Якщо потрібен якийсь шар між базовою бібліотекою мови, його можна надати, але, якщо можливо, його слід зменшити до нуля. У деяких випадках деякі частини базової бібліотеки можна замінити кращою альтернативою. N2O має надати клієнтську бібліотеку-компаньйон, зазвичай реалізовану на іншому наборі мов клієнта: JavaScript, Swift, Kotlin. Якщо ви все зробили правильно, значення N2O не повинно перевищувати 500 LOC на будь-якій мові.

[8] `https://n2o.dev/ua/deps/nitro/index.html`

Розділ 5

Система вищих мов

Присвячується автору MLTT

Перу Мартіну-Льофу

П'ятий розділ описує розвиток концептуальної моделі системи доведення теорем як сукупності: послідовності формальних мов програмування, кожна наступна з яких, складніша за попередню, має свою операційну семантику, та наслідує усі властивості попередніх мов послідовності.

Вступне слово

Батьком всіх сучасних теорій типів, зокрема HoTT, в яких працюють сучасні формальні математики можна вважати Пера Мартіна-Льофа. В його теорії мова ділиться на типи, кожен з яких визначається 4 правилами: конструктори та деконструктори, та рівняння які визначають як відбуваються обчислення та гарантують унікальність. Таке кодування природнім чином відповідає кодуванню ізоморфізмів та відображає глибинну категорну семантику типових систем. Ця робота цілком відповідає моделі типів Мартіна-Льофа та показує те тільки конфігурацію конкретного спектра мов, але і визначає модель для опису цього спектру.

5.1 Чиста система типів PTS

IEEE[1] стандарт та регуляторні документи ESA[2] визначають інтрументи та підходи до виробничого процесу верифікації та валідації. Найбільш розвинені та потужні засоби вимагають застосування математичних мов та

[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

нотацій. Ера верифікованої математики була започаткована верифікатором AUTOMATH[16] (де Брейн) розробленого під керівництвом де Брейна, а також розвиток теорії типів Мартіна-Льофа[17]. Сьогодні ми маємо Lean, Coq, F*, Agda мови які використовують числення конструкцій, Calculus of Constructions[18] (CoC) та числення індуктивних типів (Calculus of Inductive Constructions[19] (CiC). Пізніше учень де Брейна, Хенк Барендрегт класифікував послаблені чисті системи типів по трьом осям та візуазізував це за допомогою лямбда-куба[20]. Чисті мови програмування вже були імплементовані раніше (Morte[3] Габріеля Гонзалеза, Henk[21] Еріка Мейера). Чисті системи типів це системи з однм Π-типом (або ще і Σ як в ECC[22], Ope), з можливими розширеннями, такими як PTS$^\infty$ зі зліченою кількістю всесвітів[23] (Сохацький), Cedile з self-типами[24][25] (Стамп, Фу), система з K-правилами[26] (Барте).

Чисті системи типів називаються PTS, CoC Henk. Нефібраційна чиста система просто-типизованого лямбда числення називається Alonzo.

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 [4]); 2) сертифіковані обчислення для інтерпретаторів; 3) платіжні системи.

5.1.1 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[17]. 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 екстрагуванню 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.

PTS синтаксиси. Мінімальне ядро з однією аксіомою сприймає декілька лямбда ситаксисів. Перший синтаксис сумісний з системою

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

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

програмування Morte[5], та походить від неї. Інший синтаксис сумісний з синтаксисом cubicaltt[6].

$$\begin{cases} \mathsf{Sorts} = \mathsf{U}.\{\mathsf{i}\},\ \mathsf{i} : \mathsf{Nat} \\ \mathsf{Axioms} = \mathsf{U}.\{\mathsf{i}\} : \mathsf{U}.\{\mathsf{inc}\ \mathsf{i}\} \\ \mathsf{Rules} = \mathsf{U}.\{\mathsf{i}\} \rightsquigarrow \mathsf{U}.\{\mathsf{j}\} : \mathsf{U}.\{\mathsf{max}\ \mathsf{i}\ \mathsf{j}\} \end{cases} \tag{5.1}$$

Мова програмування Henk — це мова з залежними типами, яка є розширенням числення конструкцій (CoC) Тері Кокана. Саме з числення конструкцій починається сучасна обчислювальна математика. В додаток до CoC, наша мова Henk 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 Барендрегта для систем з чистими типами:

Проміжна мова чистої системи типів Henk базується на мові Henk[21], 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 Henk 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 Henk, можливо імплементовоної на іншій мові програмування та вбудованої в основну систему.

5.1.2 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

[5] http://github.com/Gabriel439/Haskell-Morte-Library
[6] http://github.com/mortberg/cubicaltt

Табл. 5.1: Перелік мов, інтерпретаторів та платформ для цільової компіляції

Ціль	**Джерело**	**Система типів**	**Застосування**
CPS	Per	π+λ+μ calculi	середовище виконання
BEAM	Hamler	System F	системна бібліотека N2O.DEV
BEAM	PTS	CoC^{∞}	фундаментальна мова
BEAM	HTS	HoTT	гомотопічна система
JavaScript	PureScript	System F	дитячі розваги
JVM	Java	F-sub[7]	реліктова історія
JVM	Scala	System F_{ω}	маргінальна промисловість
CLR	F#	System F_{ω}	маргінальна промисловість
GHC	Haskell	System F_{ω}	avoid success at all costs
GHC	Morte	CoC	фундаментальна мова Кокана
JavaScript	Kind	Self-Types + IntNet	сучасна фундаментальна мова
GHC,OCaml	Coq	CiC	промисловий прувер

```
I := list nat
U := U + 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 $\mathsf{O}_{\mathsf{PTS}}$ або O_{Π} який може вмістити цей синтаксис містить всі правила виводу внутрішньої мови категорії.

```
def PTS (lang: U) : U
 := inductive { star              (n: nat)
              | var    (x: name) (l: nat)
              | pi     (x: name) (l: nat) (d c: lang)
              | lambda (x: name) (l: nat) (d c: lang)
              | app                        (f a: lang)
              }
```

Всесвіти

Мова PTS^{∞} – це лямбда числення з залежними типами вищого порядку, розширення числення конструкцій Кокана, або системи P_{ω} Барендрегта, з предикативною (імпредикативною) ієрархією індексованих всесвітів. Це розширення мотивоване консистентністю[27] в залежній теорії типів та неможливістю кодування парадоксів Жирара-Хуркенса-Рассела[8]. Також

[8]Так званий парадокс голяра який виникає в системах $\mathsf{U} : \mathsf{U}$

для забезпечення консистентності в мові PTS відсутня аксіома Fixpoint.

$$U_0 : U_1 : U_2 : U_3 : ...$$

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

$$\frac{o : Nat}{U_o} \tag{S}$$

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 Барендрегта. 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.

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

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

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.

$$\frac{}{\Gamma : Ctx} \tag{Ctx-formation}$$

$$\frac{\Gamma : Ctx}{\emptyset : \Gamma} \tag{Ctx-intro$_1$}$$

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

5.1.3 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.

Доведення цих теорем дано в модулі базової бібліотеки розділу 3. Також можна повитися на інші доведення [20]. 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 чисту систему типів PTS (з бескінечною кількістю всесвітів) для завантаження файлів з локального довіреного сховища додається remote конструктор в синтаксичне дерево. Рекурсія по цьому конструктору заборонена.

Індекси де Брейна діють локально в межах одного імені. При додаванні існуючого імені в контекст збільшується індекс цього імені. Таким чином PTS верифікатор чистої системи типів відрізняється від канонічного приклада алгоритма верифікації CoC[18]. Він включає наступні функції мовної категорії: підстановка, зсув імені, нормалізація термів, рівність за визначенням та верифікація.

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[25][24].

```
type (:star,N)      D → (:star,N+1)
     (:var,N,I)     D → :true =proplists:is_defined N B, om:keyget N D I
     (: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)
```

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)
   (: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)
```

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
    (: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)
     (: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
   (: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))
```

Обмеження

Обмеження: 1) неможливість визначити рекурсію та індукцію без fix-point 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).

5.1.4 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
```

5.1.5 Кодування Бома

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

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

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

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

```
def list (A: U) : U
 := inductive { cons (x: A) (cs: list A)
              | nil
              }
```

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

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

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

5.1.6 Поліноміальні функтори

Існує два види формальної рекурсії: 1) перша з найменшою нерухомою точкою (як $\mathsf{F}_A(X) = 1 + A \times X$ або $\mathsf{F}_A(X) = A + X \times X$), 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 $\mathsf{F}_A(X) = A \times X$) — 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: $\nu\ X \to 1 + A \times X$
Скінченне 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 Π-типу. Таке кодування називається кодуванням Бома-Беррардуччі, а просто кодування Бома. Воно дозволяє кодувати індуктивні типи даних Π-типами чистими функціями. Проте як було показано Жеверсом[28] 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.

Екстракти

Мова Henk передбачає автоматичну генерацію сертифікованих програм в цільові платформи. Сертифікація полягає у візуальному доведенню одніє 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 працює функція екстракту в Erlang з системи типів PTS$^{\infty}$. Erlang-версія Henk 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.

Інтерпретатори

З практичної точки зору мова Henk є способом використання залежних типів та специфікації, побудовані за їх допомоги на мові Erlang. Завдяки глибокій інтеграції з Erlang вдалося мімізувати імплементацію системи до 300 рядків. Екстракт в інтерпретатор $\mathrm{O}_{\mathrm{PTS}}$ (чи інші) є альтернативною опцією для Henk. Також, мова Henk може бути легко портована на інші мови.

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 велична терму-доведення будь-яких властивостей.

FPGA

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

5.2 Система типів для W-індукції MLTT-80

Традиційно індуктивні типи входять в операційну семантику систем побудованих та основі MLTT-80 (таких як кубічні системи, або система HTS). Оригінально Мартін-Льоф виразив індуктивні дерева через W-типи, для яких також потрібно 0-тип, 1-тип, та 2-тип. Така система є менш потужною ніж система типів Крістін Полен, оскільки не дозволяє виразити загальні схеми індукції та взаємну рекурсію. Хоча з іншого боку не потребує термінейшин чекера, стріктлі позітів чекера, та взаємної рекурсивності, що дозволяє доводити семантику такого мовного ядра значно простіше.

$$\frac{A : \mathsf{Type} \quad x : A \quad B(x) : \mathsf{Type}}{W(x : A) \to B(x) : \mathsf{Type}} \tag{W-formation}$$

$$\frac{a : A \quad t : B(a) \to W}{\mathsf{sup}(a, t) : W} \tag{W-intro}$$

$$\frac{\begin{array}{l} w : W \vdash C(w) : \mathsf{Type} \\ x : A,\ u : B(x) \to W, \\ v : \Pi(y : B(x)) \to C(u(y)) \vdash c(x, u, v) : C(\mathsf{sup}(x, u)) \end{array}}{w : W \vdash \mathsf{wrec}(w, c) : C(w)} \tag{W-elim}$$

$$\frac{\begin{array}{l} w : W \vdash C(w) : \mathrm{Type} \\ x : A,\ u : B(x) \to W, \\ v : \Pi(y : B(x)) \to C(u(y)) \vdash c(x, u, v) : C(\mathsf{sup}(x, u)) \end{array}}{\begin{array}{l} x : A,\ u : B(x) \to W \vdash \mathsf{wrec}(\mathsf{sup}(x, u), c) \\ \quad = c(x, u, \lambda(y : B(x)), \mathsf{wrec}(u(y), c)) : C(\mathsf{sup}(x, u)) \end{array}} \tag{W-β}$$

Система типів з W-індукцією називається MLTT або Per.

5.3 Система індуктивних схем CIC

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. Системи типів з загальною індукцією називаються CIC, Frank або Christine.

Синтаксис

```
file -> 'module' id 'do' content : {module,'2','4'} .
content -> 'empty' : [] .
content -> line content : ['1'|'2'] .
line -> 'import' id : {import,'2'} .
line -> 'def' id tele ':' exp ':=' exp : {def,'2','5','3','7'} .
tele -> 'empty' : [] .
tele -> optic tele: ['1'|'2'] .
optic -> '(' vars ':' exp ')' : {'()','2','4'} .
vars -> id : '1' .
vars -> id vars : ['1'|'2'] .
exp -> id : '1' .
exp -> id '.' exp : {'.','1','3'} .
exp -> exp exp : {app,'1','2'} .
exp -> '(' exp ')' : '2' .
exp -> forall tele ',' exp : {'Π','2','4'} .
exp -> sigma tele ',' exp : {'Σ','2','4'} .
exp -> exp arrow exp : {'↦','1','3'} .
exp -> lam tele ',' exp : {'λ','2','4'} .
exp -> '?' : {hole} .
exp -> U : {'U'} .
exp -> app : '1' .
exp -> 'inductive' '{' sum '}' : {inductive,lists:flatten(uncons('3'))} .
constructor -> id tele : {element,'1','2'} .
constructor -> id tele ':' exp : {interval,'1','2','4'} .
sum -> 'empty' : [] .
sum -> constructor : '1' .
sum -> constructor '|' sum : ['3'|'1'] .
app -> exp exp : {app,'1','2'} .
```

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

5.4 Гомотопічна система типів HTS

Гомотопічна система типів, принаймі без вищих індуктивних типів, які є функтором на інших мовних категорій, формально моделюється досніповою (предпучковою) семантикою на категорії де Моргана $\mathrm{I} : \square_{\mathrm{n}}^{\mathrm{op}} \to \mathsf{Set}$. Так як тут теж є своя аплікація та лямбда, то можна сказати, що це ще одна лямбда система, але для безпосередньох маніпуляції багатовимірними кубами, використовуючи при цьому логіку де Моргана. Система типів з гомотопічним інтервалом називається CCHM, HTS або Anders.

Синтаксис

Тут подано повний синтаксис разом з вищими індуктивними типами.

```
start <Module.file> file
start <Module.command> repl
repl : COLON IDENT exp1 EOF | COLON IDENT EOF | exp0 EOF | EOF
file : MODULE IDENT WHERE line* EOF
path : IDENT
line : IMPORT path+ | OPTION IDENT IDENT | declarations
ident : IRREF | IDENT
lense : LPARENS ident+ COLON exp1 RPARENS
telescope : lense telescope
params : telescope | []
declarations :
  | DEF IDENT params DEFEQ exp1
  | DEF IDENT params COLON exp1 DEFEQ exp1
  | AXIOM IDENT params COLON exp1
face : LPARENS IDENT IDENT IDENT RPARENS
part : face+ ARROW exp1
exp0 : exp1 COMMA exp0 | exp1
exp1 : LSQ separated(COMMA, part) RSQ
  | LAM telescope COMMA exp1      | PI telescope COMMA exp1
  | SIGMA telescope COMMA exp1    | LSQ IRREF ARROW exp1 RSQ
  | LT ident+ GT exp1             | exp2 ARROW exp1
  | exp2 PROD exp1                | exp2
exp2 : exp2 exp3 | exp2 APPFORMULA exp3 | exp3
exp3 : LPARENS exp0 RPARENS LSQ exp0 MAP exp0 RSQ
  | HOLE                | PRE             | KAN             | IDJ exp3
  | exp3 FST             | exp3 SND        | NEGATE exp3     | INC exp3
  | exp3 AND exp3        | exp3 OR exp3    | ID exp3         | REF exp3
  | OUC exp3             | PATHP exp3      | PARTIAL exp3    | IDENT
  | IDENT LSQ exp0 MAP exp0 RSQ           | HCOMP exp3
  | LPARENS exp0 RPARENS                  | TRANSP exp3 exp3
```

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.

5.5 Структура верифікатора

На відміну від одноаксіоматичного верифікатора Per, який містить тільки один індексований всесвіт U_i, рівність за визначенням для примітивів єдиного Π-типу, та функцію верифікації τ = **type**, верифікатори Per і Anders містять додатково Σ-тип для контекстів та телескопів, більш деталізовану функцію типізації τ, та багато інших досніпових модулів, крім Π-типу, але які теж підпорядковуються системі типів Мартіна-Льофа.

Космос $\mathbb{N}$-індексованих всесвітів ω

В теорії типів всі сигнатури всіх типів живуть в ієрархіях всесвітів індексованих натуральними числами. Множина таких ієрархій називається космосом. В імплементаціях $\mathbb{N}$ завжди реалізовано як Big Integer. Верифікатор **Anders** має космос, що складається з двох ієрархій всесвітів $\omega = \{\mathsf{V}_\mathsf{i}, \mathsf{U}_\mathsf{i}\}$.

Рівність = з точністю до α-β конверсій

Рівність за визначенням (інтенсіональна) двох термів означає, що за допомогою серії альфа та бета перетворень можна довести що терми дорівнюють посимвольно (бінарно). Саме ця функція повинна бути імплементована для всіх типів у верифікаторі. Програми, які доводять рівність двох термів в теорії самого верифікатора за допомогою конструкторів рефлексивності називаються екстенсіональними рівностями. Якшо це відбувається для =-типу у всесвітах V_i — це називаються пропрозиційними рівностями, а якшо в інших ідентифікаційних системах у всесвітах U_i – називаються типовими рівностями. Інтерналізація інтенсіональної рівності за визначенням всередині теорії за допомогою Π-типів називається рівністю Лейбніца і є виводимою у всіх фібраційних верифікаторах.

Функція верифікації τ

Головна функція верифікації розпадається на систему взаємозалежних функцій $\tau = \{\mathbf{infer}, \mathbf{app}, \mathbf{check}, \mathbf{act}, \mathbf{conv}, \mathbf{eval}\}$, які повинні бути імплементовані для кожного типу, вбудованого в верифікатор.

Контексти та телескопи Σ

В теорії типів контексти, як алгебраїчні послідовності які містять сигнатури, які теж у свою чергу складаються з послідовністей пар, що складаються з імені змінної та її типу, визначаються Σ-типами.

Досніпові модулі $\mathcal{P}$ вбудованих типів

Кожен досніповий модуль повинен бути представлений у вигляді п'яти синтаксичних примітивів: 1) формації; 2) конструкції; 3) елімінації; 4) обчислювальності; 5) унікальності. Ці примітиви повинні бути узгоджені в сенсі Мартіна-Льофа та представлені у цій статті, як документація на бібліотеку верифікатора, як у тому числі дає формальне визначення примітивам в конкретній теорії $\int = \{\Pi, \Sigma, =, \mathbf{W}, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{Path}, \mathbf{Glue}\}$.

5.6 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 $\mathsf{e}_\mathsf{i} : \mathsf{O}_{\mathsf{n}+1} \to \mathsf{O}_\mathsf{n}$ є екстракторами, які понижають систему типів, при чому $\mathsf{O}_{\mathsf{CPS}} = \mathsf{O}_0$.

Базова бібліотека мови Ерланг у яку проводиться основний естракт, поставляється з дистрибутивом Erlang/OTP. Базова бібліотека $\mathsf{O}_{\mathsf{PTS}}$ наведена в репозиторії Github[9]. Гомотопічна базова бібліотека відповідає термінальній мові $\mathsf{O}_{\mathsf{CCHM}}$, та теж відкрита на Github[10]. Останні два розділи присвячені математичному моделюванню математики на цій мові.

[9] `https://github.com/groupoid/henk`
[10] `https://github.com/groupoid/anders`

Розділ 6

Бібліотека вищих мов

Присвячується вчителям
американської школи
формальної філософії та
авторам HoTT

У попередньому п'ятому 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.

Вступне слово

6.1 Інтерналізація теорії типів

Кожна мовна імплементація повинна бути протестована. Один з можливих сценаріїв тестування типових верифікаторів це пряме вбудовування в модель теорії типів виконуючого верифікатора. Так як всі типи в теорії формулюються за допомогою п'яти правил: формації, інтро, елімінації, обчисленя, рівності; ми зконструювали номінальні типи-синоніми для виконуючого верифікатора та довели, що це є реалізацією MLTT. Це може розглядатися як універсальний тест для імплементації типового верифікатора, позаяк компенсаця інтро правила та правила елімінатора пов'язані в правилі обчислення та рівності (бета та ета редукціях). Таким чином, доводжуючи реалізацію MLTT, ми доводимо властивості самого виконуючого верифікатора. MLTT-75 розкладається у спектр Π, Σ, та = типів. У цьому розділі ми побудуємо мінімальну кубічну систему необхідну для вбудовування MLTT-75 у саму себе.

Більш формально, кубічне MLTT вбудовування конструктивно виражає J елімінатор типу-рівності та його рівняння — правило обчислення, що було

неможливо до кубічної інтерпретації. Також цей розділ відкриває серію параграфів присвячених формалізації основ математики у кубічній теорії типів, MLTT моделюванню та кубічної верифікації. Так як не всі можуть бути знайомі з теорією типів, цей розділ також містить їх інтерпретації з точки зору різних розділів математики.

Додамо, що це тільки вхід в техніку прямого вбудовування і після MLTT моделювання, ми можем піднятися вище — до вбудовування в систему індуктивних типів, і далі, до вбудовування CW-комплексів як зклейок вищих індуктивних типів.

Теорія типів

Теорія типів — це універсальна мова програмування чистої математики (для доведення теорем), яка може містити довільну кількість консистентних аксіом, впорядкованих у вигляді псевдо-ізоморфізмів: 1) сигнатури типу або формації; 2) функції encode, способи конструювання елементів типу або конструкція; 3) функції decode, залежні елімінатори принципу індукції типу або елімінація; 4) рівняння бета правила або обчислювальності; 5) рівняння ета правила або унікальності. Таке визначення було дано Мартіном-Льофом, від чого теорія типів носить його ім'я MLTT.

Головна мотивація гомотопічної теорії типів — надати обчислювальну семантику гомотопічним типам та CW-комплексам. Головна ідея гомотопічної теорії [1] полягає в поєднанні просторів функцій, просторів контекстів і просторів шляхів таким чином, що вони утворюють фібраційну рівність яка збігається (доводиться в самій теорії) з простором шляхів.

Завдяки відсутності ета-правила у рівності, не кожні два доведення одного простору шляхів дорівнюють між собою, отже простір шляхів утворює багатовимірну структуру інфініті-групоїда.

Кожен тип в MLTT описується за допомогою пяти правил: 1) правила формації або сигнатура типу; 2) множина конструкторів за допомогою яких рекурсивно будують елементи типу певної сигнатури; 3) залежний елімінатор принципу індукції для цього типу; 4) бета-рівняння або правило обчислення; 5) ета-рівняння або принцип унікальності.

Найбільш цікаві — Id типи, які були додані в теорію типів в 1984 році [1] у той час як оригінальна теорія була представлена [2] в 1972 році. Предикативна ієрархія всесвітів була додана [3] в 1975.

MLTT з Id типами зберігає властивість стягуваного простору усіх доведень (uniquness of identity proofs, UIP), або ета-правило Id типа, але HoTT відхиляє UIP — ета правило не виконується, а натомість вводиться простір шляхів — Path тип [4] — так звана ∞-Groupoid інтерпретація.

Простори шляхів є ключовими для доведення властивостей обчислюваності та унікальності (бета та ета правил). Також в цій роботі ми покажемо

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

Табл. 6.1: Інтерпретації теорії типів які відповідають математичним теоріям

Теорія типів	**Логіка**	**Теорія категорій**	**Теорія гомотопій**
A тип	клас	об'єкт	простір
isProp A	твердження	(-1)-обрізаний об'єкт	простір
a:A програма	доведення	узагальнений елемени	точка
$B(x)$	предикат	індексований об'єкт	розшарування
$b(x) : B(x)$	умовне доведення	індексовані елементи	секції
$\emptyset$	$\bot$ неправда	термінальний об'єкт	пустий простір
$\mathbf{1}$	$\top$ істина	ініціальний об'єкт	сінглтон
$A + B$	$A \vee B$ діз'юнкція	кодобуток	простір кодобутків
$A \times B$	$A \wedge B$ кон'юнкція	добуток	простір добутків
$A \to B$	$A \Rightarrow B$	внутрішній Hom	простір функції
$\sum x : A, B(x)$	$\exists_{x:A} B(x)$	залежна сума	повний простір
$\prod x : A, B(x)$	$\forall_{x:A} B(x)$	залежний добуток	простір секцій
$\mathbf{Path}_A$	еквівалентність $=_A$	об'єкт типу шляхів	тип шляхів A^I
факторизація	клас еквівалентн.	фактор	фактор
W-тип	індукція	коліміт	комплекс
тип типів	всесвіт	класифікаторо об'єктів	всесвіт
квантова схема	граф доведення	струнна діаграма	

мінімальну систему де можна довести всі властивості MLTT теорії типів.

6.1.1 Структура бібліотеки

Основи

Перша частина базової бібліотеки — модальні унівалентні MLTT основи, що розділені на три групи. Перша група містить класичні типи MLTT системи описані Мартіном-Льофом, які присутні у мовах Per та Anders. Друга група містить унівалентні ідентифікаційні системи мови Anders. Третя група містить модальності мови Anders, які використовуються в диференціальній геометріїї та в теорії гомотопій. Основи пропонують фундаментальний базис який використовується для формалізації сучасної математики в таких системах доведення теорем як: Coq, Agda, Lean.

— Фібраційні
— Унівалентні
— Модальні

Математики

Друга частина базової бібліотеки Anders містить формалізації математичних теорій з різних галузей математики: аналіз, алгебра, геометрія, теорія гомотопій, теорія категорій.

Слухачам курсу (10) пропонується застосувати теорію типів для доведення початкового але нетривіального результу, який є відкритою проблемою в теорії типів для однеї із математик, що є курсами на кафедрі чистої математики (КМ-111):

— Функціональний аналіз
— Гомологічна алгебра
— Диференціальна геометрія
— Теорія гомотопій
— Теорія категорій

Програми

Третя частини базової бібліотеки, присутня у мовах Anders та Per, присвячена прикладам з промислового програмування в області автоматизації підприємств та інформаційних технологій, а саме для специфікації програмних інтерфейсів.

— Формалізація двонаправленого тракту
— Формалізація графічного веб інтерфейсу
— Формалізація бази даних з єдиним простором ключів
— Формалізація реляційної бази даних
— Формалізація системи управління процесами

Філософії

З сучасників формальною філософією в HoTT загалом займається Девід Корфілд, а формалізацією свідомості як окремий предмет вивчають Хенк Барендрегт та Горо Като. Формальна теорія природніх мов теж формалізується за допомогою MLTT, а основні теореми доводять в HoTT. В четвертій частині базової бібліотеки Anders наводяться приклади програм, які маніфестують висловлювання і теореми з формальної філософії про пустотність всіх феноменів та синтаксис, морфологію і семантику природньої української мови.

— Формалізація Мадг'яміки
— Формалізація української мови в кванторах

6.1.2 Інтерпретації теорії типів

Використовуючи теорію типів як мову, можливо закодувати на ній як різні математичні теорії, так і використовувати ці математичні теорії як взаємозамінювані фундаменти математики. Така взаємозамінюваність, яка показана в таблиці 6.1 говорить про те, що і теорія типів і математичні теорії є інтерпретаціями однієї конструкції.

Тут ми будемо говорити про наступні інтерпретації: 1) теоертико-типову; 2) категоріальну; 3) теоретико-множинну; 4) гомотопічну; 5) фібраційну або геометричну.

Реалізовуючи компаративістику інтерпретацій розглянемо на прикладах перші та основні типи теоретико-типової MLTT системи (Π, Σ, Id) зразу з точки зору декількох інтерпретацій: 1) логічної або осучасненої теоретико-типової інтерпретації, похідних системах від MLTT; 2) категоріальної або топосо-теоретичної фібраційної геометричної інтерпретації; 3) гомотопічної або кубічної інтерпретації.

Теоретико-множинна інтерпертація

Теоретико-множинна інтерпретація не зображена в таблиці 6.1, тому що є домінуючою робочою інтерпретацією фундамента сучасної математики.

Теортико-множинна інтерпретація може замістити логіку перших порядків, але не може безпосередньо оперувати вищими рівностями. Індуктивні типи в терію множин повинні вбудовуватися додатково, а саме визначення множини моделюється в гомотопічній теорії як n-тип. Наявність принципу унікальності доведень рівності (UIP) в типовій системі вже означає теоретико-множинну інтерпретацію. Не зважаючи на це теореми теорії множин без зусиль вбудовуються в будь-яку з інтерпретацій.

Логічна або теоретико-типова інтерпретація

Виходячи з засад MLTT теорії типів, кожен тип або сигнатура визначається п'ятьома правилами: 1) формації або типової сигнатури; 2) представлення або інтро-правила; 3) елімінації або загальний принцип залежної індукції; 4) правила обчислення або бета-правила; 5) принципу унікальності або ета-правила.

Формальна репрезентація всіх правила MLTT теорії буде надана в кінці цього розділу. Є загальновідомим той факт, що класичні логіки можуть бути вбудовані в інтуіціоністичну пропозиційну логіку (intuitionistic propositional logic, IPL), яка безпосередньо вбудована та є природнім (інтуіціоністичним) продовженням MLTT.

Хоча класично-логічна інтерпретація відрізняється від модернової теоретико-типової інтерпретації, вони можуть бути об'єднані як загальні логічні, тому що всі є формами механізовуваних мов програмування.

Категоріальна або топосо-теоритична інтерпретація

Категоріальна інтерпретація розповідає про теорію типів як внутрішню мову декартово-замкнених категорій та їх функторів. Головним результатом категорної інтерпретації можна назвати спряженість функторів Π та Σ, які є носіями відповідних MLTT типів та формують собою локальну декартово-замкнену категорію, яка буде надана у розділі 7.

Топосо-теоретична інтерпретація з досніповими моделями теорії типів Кокана, де фібрації конструюються як функтори, є сучасним математичним апаратом аналізу не тільки залежних але і кубічних теорій.

Гомотопічна інтерпретація

В класичних системах MLTT-72, MLTT-75 та MLTT-80 правило унікальносьі для Id типу виконується строго. Однак в гомотопічній інтерпретації просторів шляхів нам потрібно виключити це правило, аби надати багатовимірну глибину поняття рівності. Групоїдна інтерпретація теорії типів розкриває цю мотивацію та її необхідність, що було показано Мартіном Хофманом та Томасом Стрейхером в 1996.

6.1.3 Типи Π, Σ, Path

Π-тип

Π-тип — це простір, що містить залежні функції, тип кодомену яких залежить від значення типу домену. Оскільки домен розшарування присутній у кожній визначеній функції Π-тип також є залежним добутком. Простори залежних функцій використовуються в теорії типів для моделювання різноманітних математичних конструкцій, об'єктів, типів або просторів та їхніх відображень: залежних функцій, неперервних відображень, етальних відображень, розшарувань, квантору узагальнення, імплікацій тощо.

Теоретико-типова інтерпретація

Як логічна система теорія залежних типів відповідає логіці вищих порядків, одна тут даються виключно правила теоретико-типової MLTT інтерпретації.

Definition 59. (Π-Формація). Π-тип визначає спосіб у який в певному всесвіті створюється простір залежних фукнцій $f : \Pi(x : A), B(x)$ з доменом в A, та кодоменом $B : A \to U_i$.

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

```
def Pi (A: U) (B: A $\rightarrow$ U): U := $\Pi$ (x: A), B(x)
```

Definition 60. (Π-Представлення). Лямбда-конструктор визначає нову лямбда-функцію в просторі залежних функцій — це називається лямбда-абстракцією і позначається як $\lambda x.b(x)$ або $x \mapsto b(x)$.

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

```
def lambda (A: U) (B: A $\rightarrow$ U) (b: Pi A B) : Pi A B := $\lambda$ (x : A), b x
def lam (A B: U) (f: A $\rightarrow$ B) : A → B :=  $\lambda$ (x : A), f x
```

Коли кодомен не є залежним від значення в домені, функції $f : A \to B$ вивчаються в моделі теорії типів, яка називається System P_ω або числення конструкцій (Caluculus of Constructions).

Definition 61. (Π-Елімінація). Загальний принцип індукції Π-типу стверджує, що якщо предикат утримується для лямбда функцій, тоді існує функція з простору функцій в простір предикатів. Частовий випадок загальної індукції Π-типу називається λ-аплікацією, яка скорочує терм шляхом рекурсивної підстановки виразу замість аргументу з подальшою нормалізацією.

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

```
def apply (A B: U) (f: A $\rightarrow$ B) (a: A) : B := f a
def app (A: U) (B: A $\rightarrow$ U) (a: A) (f: Pi A B): B a := f a
```

Definition 62. (Композиція функцій). Композиція використовує аплікацію відповідних сигнатур.

```
def $\circ^T$ (a b c: U) : U := (b → c) → (a → b) → (a → c)
def $\circ$ (a b c : U) : $\circ^T$ a b c := $\lambda$ (g: b →
c) (f: a → b) (x: a), g (f x)
```

Theorem 1. (Π-Обчислюваність). β-правило лямбда числення, або рівняння обчислювальності показує, що композиція lam ∘ app може бути скорочена.

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

```
def $\Pi$-$\beta$ (A : U) (B : A → U) (a : A) (f : Pi A B)
  : Path (B a) (Π-apply A B (Π-lambda A B f) a) (f a)
 := idp (B a) (f a)
```

Theorem 2. (Π-Унікальність). η-правило лямбда числення, або рівняння унікальності показує, що композиція app ∘ lam може бути скорочена.

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

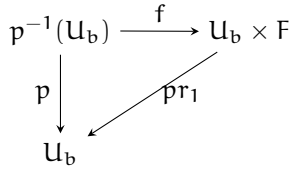
```
def $\Pi$-$\eta$ (A : U) (B : A → U) (a : A) (f : Pi A B)
  : Path (Pi A B) f (λ (x : A), f x)
 := idp (Pi A B) f
```

Гомотопічна інтерпретація

Геометрично, Π-тип — це простір секцій, у той час коли кодомен — це простір розшарувань. Лямбда-функції — це простір секцій, або точки в цих просторах, у той час, коли результат функції — це розшарування. Π-тип також репрезентує декартовий добуток сім'ї множин, узагальнюючи звичайний декартовий добуток множин.

Definition 63. (Розшарування). Розшарування відображення $p : E \to B$ у точку $y : B$ є всіма точками $x : E$, такими, що $p(x) = y$.

Definition 64. (Пучок розрашувань). Пучок розшарувань $F \to E \xrightarrow{p} B$ над тотальним простором E з розшаруванням F і базою B — це структура (F, E, p, B), де $p : E \to B$ — це сур'єктивне відображення з наступними властивостями: для будь-якої точки $y : B$ існує окіл U_b для якого гомеоморфізм $f : p^{-1}(U_b) \to U_b \times F$ робить наступну діаграму комутативною.



Definition 65. (Декартовий добуток сімейства над B). Декартовй добуток F над сімейством B — це розшарування секцій пучка з елімінюючим відображенням $app : F \times B \to E$, таким, що

$$F \times B \xrightarrow{app} E \xrightarrow{pr_1} B \tag{6.1}$$

pr_1 — це перша проекція добутка, таким чином pr_1, app — це морфізми слайс категорії $Set_{/B}$. Універсальна властивість відображень розшарування F: для всіх A і морфізму $A \times B \to E$ в $Set_{/B}$ існує унікальне відображення $A \to F$, таке, що робить все комутуючим. Таким чином, категорія з залежними добутками — це категорія зі всіма пулбеками.

Definition 66. (Тривіальний пучок розшарувань). Коли тотальний простір E є декартовим добутком $\Sigma(B, F)$ і $p = pr_1$, тоді такий пучок розшарувань називається тривіальним $(F, \Sigma(B, F), pr_1, B)$.

Theorem 3. (Тривіальний пучок розшарувань дорівнює сімейству множин). 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)
```

Категоріальна інтерпретація

Спряження Π і Σ не єдині які можуть представлені в системі.

Definition 67. (Залежний добуток). 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.

Definition 68. (Простір шарів розшарувань). Нехай $\mathbf{H}$ — це $(\infty, 1)$-топос, а $E \to B : \mathbf{H}_{/B}$ — розшарування в $\mathbf{H}$, тобто об'єкт у зрізаному топосі. Тоді простір перерізів $\Gamma_\Sigma(E)$ цього розшарування є залежним добутком:

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

Theorem 4. (Множина морфізмів). Якщо область значень є множиною, тоді простір перерізів є множиною.

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

Theorem 5. (Стягуваність). Якщо область визначення і область значень є стягуваними, то простір перерізів також є стягуваним.

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

Definition 69. (Шар розшарування). Перерізом морфізму $f : A \to B$ у деякій категорії називається морфізм $g : B \to A$ такий, що їхня композиція $f \circ g : B \xrightarrow{g} A \xrightarrow{f} B$ дорівнює тотожному морфізму на B.

Σ-тип

Σ — це залежний тип суми, узагальнення добутків. Тип Σ є повним простором розшарування. Елемент повного простору формується як пара, що складається з базової точки та розшарування.

Теоретико-типов інтерпретація

Definition 70. (Σ-Формація).Тип Σ утворює залежну суму для базового типу A та залежного типу B. Це правило формування типу в теорії типів, де $\mathcal{U}$ — універсум типів. Задане $A : \mathcal{U}$ та $B : A \to \mathcal{U}$ (функція, що кожному $a : A$ ставить у відповідність тип $B(a)$, тип $\Sigma(A, B)$ є сукупністю всіх пар $(x, B(x))$, що описують залежність між базою та розшаруванням.

```
def Sigma (A : U) (B : A -> U) : U := Σ (x : A), B x
```

Definition 71. (Σ-Предствалення). Пара (a, b) є конструктором для типу $\Sigma(A, B)$. Конструктор $\Sigma(A, B)$ через пару (a, b), де $b : B(a)$. Це базовий спосіб введення елементів у залежний тип суми.

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

Definition 72 (Σ-Елімінація)**.** Ці правила визначають проєкції та індукцію для типу Σ.

```
def pr1 (A: U) (B: A -> U) (x: Sigma A B): A := x.1
def pr2 (A: U) (B: A -> U) (x: Sigma A B): B (pr1 A B x) := x.2
def 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
```

Theorem 6 (Σ-Обчислюваність)**.** Теорема стверджує, що проєкції pr_1 і pr_2, застосовані до сконструйованої пари (a, b), повертають відповідно a та b. Equ позначає еквівалентність у термінах теорії типів, гарантуючи коректність обчислень.

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

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

Theorem 7. (Σ-Унікальність). Теорема стверджує, що будь-який елемент $p : \Sigma(A, B)$ еквівалентний парі, складеній із його проєкцій $(\mathrm{pr}_1, \mathrm{pr}_2)$, Це відображає принцип унікальності в теорії типів, де структура пари повністю визначається її компонентами.

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

Категоріальна інтерпретація

Definition 73. (Залежна сума). 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.

Теоретико-множинна інтерпретація

Theorem 8. (Аксіома вибору). 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))
```

Theorem 9. (Повний простір). 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))
```

Theorem 10. (Σ-Стягуваність). 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)
```

Theorem 11. (Шлях між сімами). 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)
```

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.

Кубічна інтерпретація

Definition 74. (Path-Формація).

```
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
```

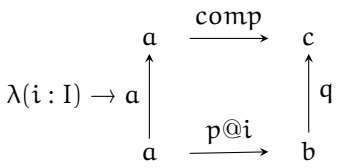
Definition 75. (Path-рефлексивність). Returns an element of reflexivity path space for a given value of the type. The inhabitant of that path space is the lambda on the homotopy interval $[0, 1]$ that returns a constant value a. Written in syntax as <i>a which equals to $\lambda\ (i : I) \to a$.

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

Definition 76. (Path-аплікація). You can apply face to path.

```
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
```

Definition 77. (Path-композиція). Composition operation allows to build a new path by given to paths in a connected point.




```
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 []
```

[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

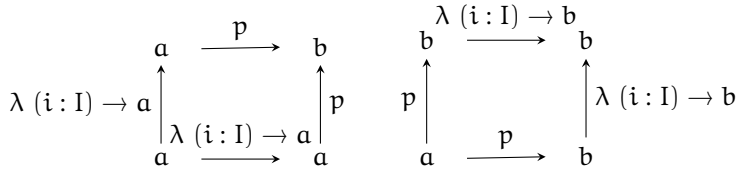
[8]Carlo Angiuli, Kuen-Bang Hou (Favonia), Robert Harper. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. 2018. https://www.cs.cmu.edu/~cangiuli/papers/ccctt.pdf

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

Theorem 12. (Path-інверсія).

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

Definition 78. (Path-з'єднання). Connections allows you to build square with given only one element of path: i) $\lambda\ (i, j : I) \to p\ @\ min(i, j)$; ii) $\lambda\ (i, j : I) \to p\ @\ max(i, j)$.




```
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)
```

Theorem 13. (Конгруентність). 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 $|\lambda$ (i:I) $\to$ 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)
```

Theorem 14. (Транспорт). 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
```

Теоретико-типова інтерпретація

Definition 79. (Сінглтон).

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

Theorem 15. (Інстанс сінглтона).

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

Theorem 16. (Стягуваність сінглтона).

```
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)
```

Theorem 17. (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)
```

Theorem 18. (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)
```

Theorem 19. (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
```

Theorem 20. (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.

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**.

6.1.4 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.

Definition 80. (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.

Definition 81. (Sorts). N-indexed set of universes $\mathrm{U}_{n\in\mathrm{N}}$. 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 $\mathcal{U}_0$ universe. Sorts represented in type checker as a separate constructor.

Definition 82. (Axioms). The inclusion rules $\mathrm{U}_i : \mathrm{U}_j, i, j \in \mathrm{N}$, 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.

Definition 83. (Rules). The set of landings $\mathrm{U}_i \to \mathrm{U}_j : \mathrm{U}_{\lambda(i,j),i,j\in\mathrm{N}}$, where $\lambda : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$. These rules define term dependence or how we land (in which universe) formation rules in definitions.

Definition 84. (Predicative hierarchy). If λ in Rules is an uncurried function $\max : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ then such universe hierarchy is called predicative.

Definition 85. (Impredicative hierarchy). If λ in Rules is a second projection of a tuple $\mathrm{snd} : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ then such universe hierarchy is called impredicative.

Definition 86. (Definitional Equality). For any $\mathrm{U}_i, i \in \mathrm{N}$ 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.

Definition 87. (SAR). The universum space is configured with a triple of: i) sorts, a set of universes $\mathrm{U}_{n\in\mathrm{N}}$ indexed over set N; ii) axioms, a set of inclusions $\mathrm{U}_i : \mathrm{U}_j, i, j \in \mathrm{N}$; iii) rules of term dependence universe landing, a set of landings $\mathrm{U}_i \to \mathrm{U}_j : \mathrm{U}_{\lambda(i,j),i,j\in\mathrm{N}}$, where λ could be function $\mathtt{max}$ (predicative) or $\mathtt{snd}$ (impredicative).

Example 4. (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 $\lambda = \mathrm{snd}$.

Example 5. (PTS$^\infty$). SAR = $\{U_{i\in N}, U_i : U_{j;i<j;i,j\in N}, U_i \to U_j : U_{\lambda(i,j);i,j\in N}\}$. Where $\mathsf{U}_\mathfrak{i}$ is a universe of $\mathfrak{i}$-level or $\mathfrak{i}$-category in categorical interpretation. The working prototype of PTS$^\infty$ is given in **Addendum I: Pure Type System for Erlang**[10].

6.1.5 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.

Definition 88. (Empty Context).

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

Definition 89. (Context Comprehension).

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

Definition 90. (Context Derivability).

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

[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

6.1.6 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.

Definition 91. (MLTT). The MLTT as a Type is defined by taking all rules for Π, Σ and Path types into one Σ telescope or context.

```
def MLTT (A: U) : U := Σ
    (Π–form   : Π (B: A → U), U)
    (Π–ctor₁  : Π (B: A → U), Pi A B → Pi A B)
    (Π–elim₁  : Π (B: A → U), Pi A B → Pi A B)
    (Π–comp₁  : Π (B: A → U) (a: A) (f: Pi A B),
                Equ (B a) (Π–elim₁ B (Π–ctor₁ B f) a) (f a))
    (Π–comp₂  : Π (B: A → U) (a:  A) (f: Pi A B),
                Equ (Pi A B) f (λ (x : A), f x))
    (Σ–form   : Π (B: A → U), U)
    (Σ–ctor₁  : Π (B: A → U) (a : A) (b : B a), Sigma A B)
    (Σ–elim₁  : Π (B: A → U) (p : Sigma A B), A)
    (Σ–elim₂  : Π (B: A → U) (p : Sigma A B), B (pr₁ A B p))
    (Σ–comp₁  : Π (B: A → U) (a : A) (b: B a),
                Equ A a (Σ–elim₁ B (Σ–ctor₁ B a b)))
    (Σ–comp₂  : Π (B: A → U) (a : A) (b: B a),
                Equ (B a) b (Σ–elim₂ B (a, b)))
    (Σ–comp₃  : Π (B: A → U) (p : Sigma A B),
                Equ (Sigma A B) p (pr₁ A B p, pr₂ A B p))
    (=–form   : Π (a: A), A → U)
    (=–ctor₁  : Π (a: A), Equ A a a)
    (=–elim₁  : Π (a: A) (C: D A) (d: C a a (=–ctor₁ a))
                  (y: A) (p: Equ A a y), C a y p)
    (=–comp₁  : Π (a: A) (C: D A) (d: C a a (=–ctor₁ a)),
                Equ (C a a (=–ctor₁ a)) d (=–elim₁ a C d a (=–ctor₁ a))), U
```

Theorem 21. (Model Check). There is an instance of MLTT.

```
theorem instance (A : U) : MLTT A :=
    (Pi A, lambda A, app A, comp₁ A, comp₂ A,
     Sigma A, pair A, pr₁ A, pr₂ A, comp₃ A, comp₄ A, comp₅ A,
     Equ A, refl A, J A, comp₆ 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.

```
$ rlwrap ./anders.native check ./experiments/mltt.anders
File loaded.
> :n instance
TYPE: Π (A : U), Σ (Π–form : Π (B : (A → U)), U), Σ (Π–ctor₁ :
      Π (B : (A → U)), (Π (x : A), (B x) → Π (x : A), (B x))),
      Σ (Π–elim₁ : Π (B : (A → U)), (Π (x : A), (B x) →
      Π (x : A), (B x))), Σ (Π–comp₁ : Π (B : (A → U)),
      Π (a : A), Π (f : Π (x : A), (B x)), Π (P : ((B a) → U)),
        ((P (((Π–elim₁ B) ((Π–ctor₁ B) f)) a)) → (P (f a)))),
      Σ (Π–comp₂ : Π (B : (A → U)), Π (a : A), Π (f :
      Π (x : A), (B x)), Π (P : (Π (x : A), (B x) → U)),
        ((P f) → (P λ (x : A), (f x)))), Σ (Σ–form :
      Π (B : (A → U)), U), Σ (Σ–ctor₁ : Π (B : (A → U)),
      Π (a : A), Π (b : (B a)), Σ (x : A), (B x)), Σ (Σ–elim₁ :
      Π (B : (A → U)), Π (p : Σ (x : A), (B x)), A),
      Σ (Σ–elim₂ : Π (B : (A → U)), Π (p : Σ (x : A), (B x)),
        (B p.1)), Σ (Σ–comp₁ : Π (B : (A → U)), Π (a : A),
      Π (b : (B a)), Π (P : (A → U)), ((P a) → (P ((Σ–elim₁ B)
        (((Σ–ctor₁ B) a) b))))), Σ (Σ–comp₂ : Π (B : (A → U)),
      Π (a : A), Π (b : (B a)), Π (P : ((B a) → U)), ((P b) →
        (P ((Σ–elim₂ B) (a, b))))), Σ (Σ–comp₃ : Π (B : (A → U)),
      Π (p : Σ (x : A), (B x)), Π (P : (Σ (x : A), (B x) → U)),
        ((P p) → (P (p.1, p.2)))), Σ (=–form : Π (a : A),
        (A → U)), Σ (=–ctor₁ : Π (a : A), Π (P : (A → U)),
        ((P a) → (P a))), Σ (=–elim₁ : Π (a : A), Π (C :
      Π (x : A), Π (y : A), (Π (P : (A → U)), ((P x)
      → (P y)) → U)), Π (d : (((C a) a) (=–ctor₁ a))),
      Π (y : A), Π (p : Π (P : (A → U)), ((P a) → (P y))),
        (((C a) y) p)), Σ (=–comp₁ : Π (a : A), Π (C :
      Π (x : A), Π (y : A), (Π (P : (A → U)), ((P x)
      → (P y)) → U)), Π (d : (((C a) a) (=–ctor₁ a))),
      Π (P : ((((C a) a) (=–ctor₁ a)) → U)), ((P d) → (P
        (((((=–elim₁ a) C) d) a) (=–ctor₁ a))))), U
EVAL: λ (A : U), (λ (B : (A → U)), Π (x : A), (B x), (
      λ (B : (A → U)), λ (b : Π (x : A), (B x)),
      λ (x : A), (b x), (λ (B : (A → U)), λ (f :
      Π (x : A), (B x)), λ (a : A), (f a), (λ (B : (A → U)),
      λ (a : A), λ (f : Π (x : A), (B x)),
      λ (P : ((B a) → U)), λ (u : (P (f a))), u, (
      λ (B : (A → U)), λ (a : A), λ (f : Π (x : A), (B x)),
      λ (P : (Π (x : A), (B x) → U)), λ (u : (P f)), u, (
      λ (B : (A → U)), Σ (x : A), (B x), (λ (B : (A → U)),
      λ (a : A), λ (b : (B a)), (a, b), (λ (B : (A → U)),
      λ (x : Σ (x : A), (B x)), x.1, (λ (B : (A → U)),
      λ (x : Σ (x : A), (B x)), x.2, (λ (B : (A → U)),
      λ (a : A), λ (b : (B a)), λ (P : (A → U)),
      λ (u : (P a)), u, (λ (B : (A → U)), λ (a : A),
      λ (b : (B a)), λ (P : ((B a) → U)),
      λ (u : (P b)), u, (λ (B : (A → U)), λ (p :
      Σ (x : A), (B x)), λ (P : (Σ (x : A), (B x) → U)),
      λ (u : (P p)), u, (λ (x : A), λ (y : A), Π (P : (A
      → U)), ((P x) → (P y)), (λ (x : A), λ (P : (A → U)),
      λ (u : (P x)), u, ((J A), ((comp₆ A), A))))))))))))))))
```

6.2 Індуктивні типи

6.2.1 Empty, Unit

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
```

6.2.2 Bool, Maybe, Either, Tuple

Definition 92. (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 }
```

Definition 93. (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 }
```

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 }
```

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
```

6.2.3 Nat, List, Stream

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)
```

Definition 94. (List). The data type of list L over a given set A can be represented as the initial algebra $(\mu L_A, \mathsf{in})$ of the functor $L_A(X) = 1 + (A \times X)$. Denote $\mu L_A = \mathsf{List}(A)$. The constructor functions $\mathsf{nil} : 1 \to \mathsf{List}(A)$ and $\mathsf{cons} : A \times \mathsf{List}(A) \to \mathsf{List}(A)$ are defined by $\mathsf{nil} = \mathsf{in} \circ \mathsf{inl}$ and $\mathsf{cons} = \mathsf{in} \circ \mathsf{inr}$, so $\mathsf{in} = [\mathsf{nil}, \mathsf{cons}]$.

```
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
```

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)
```

6.2.4 Fin, Vector, Seq

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
```

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)
```

6.2.5 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 [29] and [30]. 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
```

6.3 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 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**.

6.3.1 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

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

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

families). The homotopy interval $\mathrm{I} = [0, 1]$ is the perfect foundation for definition of homotopy.

Definition 95. (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 $\mathsf{i0}, \mathsf{i1} : \mathbf{I}$ to $x, y : A$ one can obtain identity or equality type from classic type theory.

Definition 96. (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 }
```

Definition 97. (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)
```

6.3.2 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].

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

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

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.

```
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.

6.3.3 Функціональна екстенсіональність

Definition 98. (funExt-Formation)

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

Definition 99. (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
```

Definition 100. (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
```

Definition 101. (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
```

Definition 102. (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
```

6.3.4 Пулбеки

Definition 103. (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
```

Definition 104. (Ядро).

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

Definition 105. (Гомотопічне розшарування).

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

Definition 106. (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)
```

Theorem 22. (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)
```

6.3.5 Пушаути та фібрації

Definition 107. (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) ]
```

Definition 108. (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)
```

Definition 109. (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)
```

Definition 110. (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)))
```

Definition 111. (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)
```

6.3.6 Еквівалентність, Ізоморфізм, Унівалентність

Definition 112. (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
```

Definition 113. (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
```

Definition 114. (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
```

Definition 115. (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
```

Definition 116. (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
```

Ізоморфізм.

Definition 117. (iso-Formation)

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

Definition 118. (iso-Introduction)

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

Definition 119. (iso-Elimination)

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

Definition 120. (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
```

Definition 121. (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
```

Унівалентність.

Definition 122. (uni-Formation)

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

Definition 123. (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)) ]
```

Definition 124. (uni-Elimination)

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

Definition 125. (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))) ]
```

Definition 126. (uni-Uniqueness)

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

6.4 Вищі індуктивні типи

CW-complexes are central to HoTT and appear in cubical type checkers as HITs. Unlike inductive types (recursive trees), HITs encode CW-complexes, capturing points (0-cells) and higher paths (n-cells). The definition of an HIT specifies a CW-complex through cubical composition, an initial algebra in the cubical model.

6.4.1 Suspension

The suspension ΣA of a type A is a higher inductive type that constructs a new type by adding two points, called poles, and paths connecting each point of A to these poles. It is a fundamental construction in homotopy theory, often used to shift homotopy groups, e.g., obtaining S^{n+1} from S^n.

Definition 127. (Formation). For any type $A : \mathcal{U}$, there exists a suspension type $\Sigma A : \mathcal{U}$.

Definition 128. (Constructors). For a type $A : \mathcal{U}$, the suspension $\Sigma A : \mathcal{U}$ is generated by the following higher inductive compositional structure:

$$\Sigma := \begin{cases} \text{north} \\ \text{south} \\ \text{merid} : (a : A) \to \text{north} \equiv \text{south} \end{cases}$$

```
def Σ (A: U) : U
 := inductive { north
              | south
              | merid (a: A) : north ≡ south
              }
```

Theorem 23. (Elimination). For a family of types $B : \Sigma A \to \mathcal{U}$, points $n : B(\text{north})$, $s : B(\text{south})$, and a family of dependent paths

$$m : \Pi(a : A), \text{PathOver}(B, \text{merid}(a), n, s),$$

there exists a dependent map $\text{Ind}_{\Sigma A} : (x : \Sigma A) \to B(x)$, such that:

$$\begin{cases} \text{Ind}_{\Sigma A}(\text{north}) = n \\ \text{Ind}_{\Sigma A}(\text{south}) = s \\ \text{Ind}_{\Sigma A}(\text{merid}(a, i)) = m(a, i) \end{cases}$$

```
def PathOver (B: Σ A -> U) (a: A) (n: B north) (s: B south) : U
  := PathP (λ i , B (merid a @ i)) n s

def IndΣA (A: U) (B: Σ A -> U) (n: B north) (s: B south)
    (m: (a: A) -> PathOver B (merid a) n s) : (x: Σ A) -> B x
 := split { north -> n | south -> s | merid a @ i -> m a @ i }
```

Theorem 24. (Computation).

$$\text{Ind}_\Sigma A(\text{north}) = n \text{Ind}_\Sigma A(\text{south}) = s \text{Ind}_\Sigma A(\text{merid}(a, i)) = m(a, i)$$

```
def Σ–β (A: U) (B: ΣA -> U) (n: B north) (s: B south)
    (m: (a: A) -> PathOver B (merid a) n s) (x: Σ A)
  : Path (B x) (Σ–I A B n s m x)
    split { north -> n | south -> s | merid a @ i -> m a @ i }
```

Theorem 25. (Uniqueness). Any two maps $h_1, h_2 : (x : \Sigma A) \to B(x)$ are homotopic if they agree on north, south, and merid, i.e., if $h_1(\text{north}) = h_2(\text{north})$, $h_1(\text{south}) = h_2(\text{south})$, and $h_1(\text{merid } a) = h_2(\text{merid } a)$ for all $a : A$.

6.4.2 Pushout

The pushout (amalgamation) is a higher inductive type that constructs a type by gluing two types A and B along a common type C via maps $f : C \to A$ and $g : C \to B$. It is a fundamental construction in homotopy theory, used to model cell attachment and cofibrant objects, generalizing the topological notion of a pushout.

Definition 129. (Formation). For types $A, B, C : \mathcal{U}$ and maps $f : C \to A$, $g : C \to B$, there exists a pushout $\sqcup(A, B, C, f, g) : \mathcal{U}$.

Definition 130. (Constructors). The pushout is generated by the following higher inductive compositional structure:

$$\sqcup := \begin{cases} po_1 : A \to \sqcup(A, B, C, f, g) \\ po_2 : B \to \sqcup(A, B, C, f, g) \\ po_3 : (c : C) \to po_1(f(c)) \equiv po_2(g(c)) \end{cases}$$

```
def ⊔ (A B C: U) (f: C -> A) (g: C -> B) : U
 := inductive { po₁ (a: A)
              | po₂ (b: B)
              | po₃ (c: C) : po₁(f(c)) ≡ po₂(g(c))
              }
```

Theorem 26. (Elimination). For a type $D : \mathcal{U}$, maps $u : A \to D$, $v : B \to D$, and a family of paths $p : (c : C) \to u(f(c)) \equiv v(g(c))$, there exists a map $\mathrm{Ind}_\sqcup : \sqcup(A, B, C, f, g) \to D$, such that:

$$\begin{cases} \mathrm{Ind}_\sqcup(po_1(a)) = u(a) \\ \mathrm{Ind}_\sqcup(po_2(b)) = v(b) \\ \mathrm{Ind}_\sqcup(po_3(c, i)) = p(c, i) \end{cases}$$

```
def PathOver (A B C : U) (f : C → A) (g : C → B)
    (D : ⊔ A B C f g → U)
    (c : C) (u : D (po₁ (f c))) (v : D (po₂ (g c))) : U
 := PathP (λ i , D (po₃ c i)) u v

def Ind⊔ : (A B C : U) (f : C → A) (g : C → B)
    (D : ⊔ A B C f g → U)
    (u : (a : A) → D (po₁ a))
    (v : (b : B) → D (po₂ b))
    (p : (c : C) → PathOver D c (u (f c)) (v (g c)))
  : (x : ⊔ A B C f g) → D x
 := split { po₁ a → u a | po₂ b → v b | po₃ c @ i → p c @ i }
```

Theorem 27. (Computation). For $x : \sqcup(A, B, C, f, g)$,

$$\begin{cases} \mathrm{Ind}_\sqcup(po_1(a)) \equiv u(a) \\ \mathrm{Ind}_\sqcup(po_2(b)) \equiv v(b) \\ \mathrm{Ind}_\sqcup(po_3(c, i)) \equiv p(c, i) \end{cases}$$

Theorem 28. (Uniqueness). Any two maps $u, v : \sqcup(A, B, C, f, g) \to D$ are homotopic if they agree on po_1, po_2, and po_3, i.e., if $u(po_1(a)) = v(po_1(a))$ for all $a : A$, $u(po_2(b)) = v(po_2(b))$ for all $b : B$, and $u(po_3(c)) = v(po_3(c))$ for all $c : C$.

Example 6. (Cell Attachment) The pushout models the attachment of an n-cell to a space X. Given $f : S^{n-1} \to X$ and inclusion $g : S^{n-1} \to D^n$, the pushout $\sqcup(X, D^n, S^{n-1}, f, g)$ is the space $X \cup_f D^n$, attaching an n-disk to X along f.

$$\begin{array}{ccc} S^{n-1} & \xrightarrow{f} & X \\ \downarrow{\scriptstyle g} & & \downarrow \\ D^n & \longrightarrow & X \cup_f D^n \end{array}$$

6.4.3 Spheres

Spheres are higher inductive types with higher-dimensional paths, representing fundamental topological spaces.

Definition 131. (Pointed n-Spheres) The n-sphere S^n is defined recursively as a type in the universe $\mathcal{U}$ using general recursion over dimensions:

$$\mathbb{S}^n := \begin{cases} \text{point} : \mathbb{S}^n, \\ \text{surface} : < i_1, ...i_n > [\ (i_1 = 0) \to \text{point}, (i_1 = 1) \to \text{point},\ ... \\ \qquad\qquad\qquad\qquad (i_n = 0) \to \text{point}, (i_n = 1) \to \text{point}\ ] \end{cases}$$

Definition 132. (n-Spheres via Suspension) The n-sphere S^n is defined recursively as a type in the universe $\mathcal{U}$ using general recursion over natural numbers $\mathbb{N}$. For each $n \in \mathbb{N}$, the type $S^n : \mathcal{U}$ is defined as:

$$\mathbb{S}^n := \begin{cases} S^0 = \mathbf{2}, \\ S^{n+1} = \Sigma(S^n). \end{cases}$$

```
def sphere : ℕ → U := ℕ-iter U 2 Σ
```

This iterative definition applies the suspension functor Σ to the base type $\mathbf{2}$ (0-sphere) n times to obtain S^n.

Example 7. (Sphere as CW-Complex) The n-sphere S^n can be constructed as a CW-complex with one 0-cell and one n-cell:

$$\begin{cases} X_0 = \{\text{base}\}, \text{ one point} \\ X_k = X_0 \text{ for } 0 < k < n, \text{ no additional cells} \\ X_n : \text{Attachment of an } n\text{-cell to } X_{n-1} = \{\text{base}\} \text{ along } f : S^{n-1} \to \{\text{base}\} \end{cases}$$

The constructor cell attaches the boundary of the n-cell to the base point, yielding the type S^n.

6.4.4 Hub and Spokes

The hub and spokes construction $\odot$ defines an $\mathfrak{n}$-truncation, ensuring that the type has no non-trivial homotopy groups above dimension $\mathfrak{n}$. It models the type as a CW-complex with a hub (central point) and spokes (paths to points).

Definition 133. (Formation). For types $S, A : \mathcal{U}$, there exists a hub and spokes type $\odot\,(S, A) : \mathcal{U}$.

Definition 134. (Constructors). The hub and spokes type is freely generated by the following higher inductive compositional structure:

$$\odot := \begin{cases} \text{base} : A \to \odot\,(S, A) \\ \text{hub} : (S \to \odot\,(S, A)) \to \odot\,(S, A) \\ \text{spoke} : (f : S \to \odot\,(S, A)) \to (s : S) \to \text{hub}(f) \equiv f(s) \end{cases}$$

```
def ⊙ (S A: U) : U
 := inductive { base (x: A)
              | hub (f: S -> ⊙ S A)
              | spoke (f: S -> ⊙ S A) (s:S) : hub f ≡ f s
              }
```

Theorem 29. (Elimination). For a family of types $P : \text{HubSpokes}\,S\,A \to \mathcal{U}$, maps $\text{pbase} : (x : A) \to P(\text{base}\,x)$, $\text{phub} : (f : S \to \text{HubSpokes}\,S\,A) \to P(\text{hub}\,f)$, and a family of paths $\text{pspoke} : (f : S \to \text{HubSpokes}\,S\,A) \to (s : S) \to \text{PathP}\,(< i > P(\text{spoke}\,f\,s\,@\,i))\,(\text{phub}\,f)\,(P(f\,s))$, there exists a map $\text{hubSpokesInd} : (z : \text{HubSpokes}\,S\,A) \to P(z)$, such that:

$$\begin{cases} \text{Ind}_{\odot}\,(\text{base}\,x) = \text{pbase}\,x \\ \text{Ind}_{\odot}\,(\text{hub}\,f) = \text{phub}\,f \\ \text{Ind}_{\odot}\,(\text{spoke}\,f\,s\,@\,i) = \text{pspoke}\,f\,s\,@\,i \end{cases}$$

6.4.5 Truncation

Set Truncation

Definition 135. (Formation). Set truncation (0-truncation), denoted $\|A\|_0$, ensures that the type is a set, with homotopy groups vanishing above dimension 0.

Definition 136. (Constructors). For $A : \mathcal{U}$, $\|A\|_0 : \mathcal{U}$ is defined by the following higher inductive compositional structure:

$$\|\_\|_0 := \begin{cases} \text{inc} : A \to \|A\|_0 \\ \text{squash} : (a, b : \|A\|_0) \to (p, q : a \equiv b) \to p \equiv q \end{cases}$$

```
def ‖_‖₀ (A: U) : U
 := inductive { inc (a: A)
              | squash (a b: ‖A‖₀) (p q: Path (‖A‖₀) a b)
                <i j> [ (i = 0) -> p @ j, (i = 1) -> q @ j,
                        (j = 0) -> a,     (j = 1) -> b ]
              }
```

Theorem 30. (Elimination $\|A\|_0$) For a set $B : \mathcal{U}$ (i.e., isSet(B)), and a map $f : A \to B$, there exists setTruncRec : $\|A\|_0 \to B$, such that $\text{Ind}_{\|A\|_0}(\text{inc}(a)) = f(a)$.

Groupoid Truncation

Definition 137. (Formation). Groupoid truncation (1-truncation), denoted $\|A\|_1$, ensures that the type is a 1-groupoid, with homotopy groups vanishing above dimension 1.

Definition 138. (Constructors). For $A : \mathcal{U}$, $\|A\|_1 : \mathcal{U}$ is defined by the following higher inductive compositional structure:

$$\|\_\|_1 := \begin{cases} \text{inc} : A \to \|A\|_1 \\ \text{squash} : (a, b : \|A\|_1) \to (p, q : a \equiv b) \to (r, s : p \equiv q) \to r \equiv s \end{cases}$$

```
def ‖_‖₁ (A: U) : U
 := inductive { inc (a: A)
              | squash (a b: ‖A‖₁) (p q: Path (‖A‖₁) a b)
                (r s: Path (Path (‖A‖₁) a b) p q) <i j k>
                [ (i = 0) -> r @ j @ k, (i = 1) -> s @ j @ k,
                  (j = 0) -> p @ k,     (j = 1) -> q @ k,
                  (k = 0) -> a,         (k = 1) -> b ]
              }
```

Theorem 31. (Elimination $\|A\|_1$) For a 1-groupoid $B : \mathcal{U}$ (i.e., isGroupoid(B)), and a map $f : A \to B$, there exists $\text{Ind}_{\|A\|_1} : \|A\|_1 \to B$, such that $\text{Ind}_{\|A\|_1}(\text{inc}(a)) = f(a)$.

6.4.6 Quotients

Set Quotient Spaces

Quotient spaces are a powerful computational tool in type theory, embedded in the core of Lean.

Definition 139. (Formation). Set quotient spaces construct a type A, quotiented by a relation $R : A \to A \to \mathcal{U}$, ensuring that the result is a set.

Definition 140. (Constructors). For a type $A : \mathcal{U}$ and a relation $R : A \to A \to \mathcal{U}$, the set quotient space $A/R : \mathcal{U}$ is freely generated by the following higher inductive compositional structure:

$$A/R := \begin{cases} \text{quot} : A \to A/R \\ \text{ident} : (a, b : A) \to R(a, b) \to \text{quot}(a) \equiv \text{quot}(b) \\ \text{trunc} : (a, b : A/R) \to (p, q : a \equiv b) \to p \equiv q \end{cases}$$

```
def / (A: U) (R: A -> A -> U) : U
  := inductive { quot (a: A)
               | ident (a b: A) (r: R a b) : quot(a) ≡ quot(b)
               | trunc (a b : / A R) (p q : Path (/ A R) a b)
                 <i j> [ (i = 0) -> p @ j , (i = 1) -> q @ j ,
                         (j = 0) -> a ,     (j = 1) -> b ]
               }
```

Theorem 32. (Elimination). For a family of types $B : A/R \to \mathcal{U}$ with $\text{isSet}(Bx)$, and maps $f : (x : A) \to B(\text{quot}(x))$, $g : (a, b : A) \to (r : R(a, b)) \to \text{PathP}(< i > B(\text{ident}(a, b, r)\ @\ i))(f(a))(f(b))$, there exists $\text{Ind}_{A/R} : \Pi(x : A/R), B(x)$, such that $\text{Ind}_{A/R}(\text{quot}(a)) = f(a)$.

Groupoid Quotient Spaces

Definition 141. (Formation). Groupoid quotient spaces extend set quotient spaces to produce a 1-groupoid, including constructors for higher paths. Groupoid quotient spaces construct a type A, quotiented by a relation $R : A \to A \to \mathcal{U}$, ensuring that the result is a groupoid.

Definition 142. (Constructors). For a type $A : \mathcal{U}$ and a relation $R : A \to A \to \mathcal{U}$, the groupoid quotient space $A//R : \mathcal{U}$ includes constructors for points, paths, and higher paths, ensuring a 1-groupoid structure.

6.4.7 Wedge

The wedge of two pointed types A and B, denoted $A \vee B$, is a higher inductive type representing the union of A and B with identified base points. Topologically, it corresponds to $A \times \{y_0\} \cup \{x_0\} \times B$, where x_0 and y_0 are the base points of A and B, respectively.

Definition 143. (Formation). For pointed types A, B : pointed, the wedge $A \vee B : \mathcal{U}$.

Definition 144. (Constructors). The wedge is generated by the following higher inductive compositional structure:

$$\vee := \begin{cases} \text{winl} : A.1 \to A \vee B \\ \text{winr} : B.1 \to A \vee B \\ \text{wglue} : \text{winl}(A.2) \equiv \text{winr}(B.2) \end{cases}$$

```
def ∨ (A : pointed) (B : pointed) : U
 := inductive { winl (a : A.1)
              | winr (b : B.1)
              | wglue : winl(A.2) ≡ winr(B.2)
              }
```

Theorem 33. (Elimination). For a type $P : A \vee B\mathcal{U}$, maps $f : A.1 \to C$, $g : B.1 \to C$, and a path $p : \text{PathOverlue}(P, f(A.2), g(B.2))$, there exists a map $\text{Ind}_\vee : A \vee B \to C$, such that:

$$\begin{cases} \text{Ind}(\text{winl}(a)) = f(a) \\ \text{Ind}(\text{winr}(b)) = g(b) \\ \text{Ind}(\text{wglue}(x)) = p(x) \end{cases}$$

```
def PathOverGlue : (P : A ∨ B → U)
    (p : P (inl (A.2))) (q : P (inr (B.2))) : U
 := PathP (λ i → P (wglue i)) p q

def Ind∨ (A B : pointed) (C : U) (f : A.1 -> C) (g : B.1 -> C)
    (p : Path C (f A.2) (g B.2))
  : A ∨ B -> C
 := split { winl a -> f a | winr b -> g b | wglue @ x -> p @ x }
```

Theorem 34. (Computation). For $z : \text{Wedge } AB$,

$$\begin{cases} \text{Ind}_\vee(\text{winl } a) \equiv f(a) \\ \text{Ind}_\vee(\text{winr } b) \equiv g(b) \\ \text{Ind}_\vee(\text{wglue } @\, x) \equiv p\, @\, x \end{cases}$$

Theorem 35. (Uniqueness). Any two maps $h_1, h_2 : \text{Wedge } AB \to C$ are homotopic if they agree on winl, winr, and wglue, i.e., if $h_1(\text{winl } a) = h_2(\text{winl } a)$ for all $a : A.1$, $h_1(\text{winr } b) = h_2(\text{winr } b)$ for all $b : B.1$, and $h_1(\text{wglue}) = h_2(\text{wglue})$.

6.4.8 Smash Product

The smash product of two pointed types A and B, denoted $A \wedge B$, is a higher inductive type that quotients the product $A \times B$ by the pushout $A \sqcup B$. It represents the space $A \times B/(A \times \{y_0\} \cup \{x_0\} \times B)$, collapsing the wedge to a single point.

Definition 145. (Formation). For pointed types A, B : pointed, the smash product $A \wedge B : \mathcal{U}$.

Definition 146. (Constructors). The smash product is generated by the following higher inductive compositional structure:

$$A \wedge B := \begin{cases} \text{basel} : A \wedge B \\ \text{baser} : A \wedge B \\ \text{proj}(x : A.1)(y : B.1) : A \wedge B \\ \text{gluel}(a : A.2) : \text{proj}(a, B.2) \equiv \text{basel} \\ \text{gluer}(b : B.2) : \text{proj}(A.2, b) \equiv \text{baser} \end{cases}$$

```
def ∧ (A : pointed) (B : pointed) : U
 := inductive { basel
              | baser
              | proj (a : A.1) (b : B.1)
              | gluel (a : A.2) : proj(a,B.2) ≡ basel
              | gluer (a : B.2) : proj(A.2,b) ≡ baser
              }
```

Theorem 36. (Elimination). For a family of types $P : \text{Smash } A\, B \to \mathcal{U}$, points pbasel : P(basel), pbaser : P(baser), maps pproj : $(x : A.1) \to (y : B.1) \to P\,(\text{proj}\, x\, y)$, and a family of paths pgluel : $(a : A.1) \to \text{pproj}(a, B.2) \equiv \text{pbasel}$, pgluer : $(b : B.1) \to \text{pproj}(A.2, b) \equiv \text{pbaser}$, there exists a map $\text{Ind}_\wedge : (z : A \wedge B) \to P(z)$, such that:

$$\begin{cases} \text{Ind}_\wedge\,(\text{basel}) = \text{pbasel} \\ \text{Ind}_\wedge\,(\text{baser}) = \text{pbaser} \\ \text{Ind}_\wedge\,(\text{proj}\, x\, y) = \text{pproj}\, x\, y \\ \text{Ind}_\wedge\,(\text{gluel}\, a\, @\, i) = \text{pgluel}\, a\, @\, i \\ \text{Ind}_\wedge\,(\text{gluer}\, b\, @\, i) = \text{pgluer}\, b\, @\, i \end{cases}$$

```
def Ind∧ (A B : pointed) (P : A ∧ B -> U)
    (pbasel : P basel) (pbaser : P baser)
    (pproj : (x : A.1) -> (y : B.1) -> P (proj x y))
    (pgluel : (a : A.1) -> PathP (<i> P (gluel a @ i)) (pproj a B.2) pbasel)
    (pgluer : (b : B.1) -> PathP (<i> P (gluer b @ i)) (pproj A.2 b) pbaser)
  : (z : A ∧ B) -> P z
 := split { basel -> pbasel | baser -> pbaser | proj x y -> pproj x y
          | gluel a @ i -> pgluel a @ i | gluer b @ i -> pgluer b @ i }
```

Theorem 37. (Computation). For a family of types $\mathsf{P} : \mathsf{A} \wedge \mathsf{B} \to \mathcal{U}$, points $\mathsf{pbasel} : \mathsf{P}(\mathsf{basel})$, $\mathsf{pbaser} : \mathsf{P}(\mathsf{baser})$, map $\mathsf{pproj} : (\mathsf{x} : \mathsf{A.1}) \to (\mathsf{y} : \mathsf{B.1}) \to \mathsf{P}(\mathsf{proj}\,\mathsf{x}\,\mathsf{y})$, and families of paths $\mathsf{pgluel} : (\mathsf{a} : \mathsf{A.1}) \to \mathrm{PathP}\,(< \mathsf{i} > \mathsf{P}(\mathsf{gluel}\,\mathsf{a}\,@\,\mathsf{i}))\,(\mathsf{pproj}\,\mathsf{a}\,\mathsf{B.2})\,\mathsf{pbasel}$, $\mathsf{pgluer} : (\mathsf{b} : \mathsf{B.1}) \to \mathrm{PathP}\,(< \mathsf{i} > \mathsf{P}(\mathsf{gluer}\,\mathsf{b}\,@\,\mathsf{i}))\,(\mathsf{pproj}\,\mathsf{A.2}\,\mathsf{b})\,\mathsf{pbaser}$, the map $\mathrm{Ind}_\wedge : (\mathsf{z} : \mathsf{A} \wedge \mathsf{B}) \to \mathsf{P}(\mathsf{z})$ satisfies all equations for all variants of the predicate P:

$$\begin{cases} \mathrm{Ind}_\wedge\,(\mathsf{basel}) \equiv \mathsf{pbasel} \\ \mathrm{Ind}_\wedge\,(\mathsf{baser}) \equiv \mathsf{pbaser} \\ \mathrm{Ind}_\wedge\,(\mathsf{proj}\,\mathsf{x}\,\mathsf{y}) \equiv \mathsf{pproj}\,\mathsf{x}\,\mathsf{y} \\ \mathrm{Ind}_\wedge\,(\mathsf{gluel}\,\mathsf{a}\,@\,\mathsf{i}) \equiv \mathsf{pgluel}\,\mathsf{a}\,@\,\mathsf{i} \\ \mathrm{Ind}_\wedge\,(\mathsf{gluer}\,\mathsf{b}\,@\,\mathsf{i}) \equiv \mathsf{pgluer}\,\mathsf{b}\,@\,\mathsf{i} \end{cases}$$

Theorem 38. (Uniqueness). For a family of types $\mathsf{P} : \mathsf{A} \wedge \mathsf{B} \to \mathcal{U}$, and maps $\mathsf{h}_1, \mathsf{h}_2 : (\mathsf{z} : \mathsf{A} \wedge \mathsf{B}) \to \mathsf{P}(\mathsf{z})$, if there exist paths $e_{\mathrm{basel}} : \mathsf{h}_1(\mathsf{basel}) \equiv \mathsf{h}_2(\mathsf{basel})$, $e_{\mathrm{baser}} : \mathsf{h}_1(\mathsf{baser}) \equiv \mathsf{h}_2(\mathsf{baser})$, $e_{\mathrm{proj}} : (\mathsf{x} : \mathsf{A.1}) \to (\mathsf{y} : \mathsf{B.1}) \to \mathsf{h}_1(\mathsf{proj}\,\mathsf{x}\,\mathsf{y}) \equiv \mathsf{h}_2(\mathsf{proj}\,\mathsf{x}\,\mathsf{y})$, $e_{\mathrm{gluel}} : (\mathsf{a} : \mathsf{A.1}) \to \mathrm{PathP}\,(< \mathsf{i} > \mathsf{h}_1(\mathsf{gluel}\,\mathsf{a}\,@\,\mathsf{i}) \equiv \mathsf{h}_2(\mathsf{gluel}\,\mathsf{a}\,@\,\mathsf{i}))\,(e_{\mathrm{proj}}\,\mathsf{a}\,\mathsf{B.2})\,e_{\mathrm{basel}}$, $e_{\mathrm{gluer}} : (\mathsf{b} : \mathsf{B.1}) \to \mathrm{PathP}\,(< \mathsf{i} > \mathsf{h}_1(\mathsf{gluer}\,\mathsf{b}\,@\,\mathsf{i}) \equiv \mathsf{h}_2(\mathsf{gluer}\,\mathsf{b}\,@\,\mathsf{i}))\,(e_{\mathrm{proj}}\,\mathsf{A.2}\,\mathsf{b})\,e_{\mathrm{baser}}$, then $\mathsf{h}_1 \equiv \mathsf{h}_2$, i.e., there exists a path $(\mathsf{z} : \mathsf{A} \wedge \mathsf{B}) \to \mathsf{h}_1(\mathsf{z}) \equiv \mathsf{h}_2(\mathsf{z})$.

6.4.9 Join

The join of two types A and B, denoted $A \vee B$, is a higher inductive type that constructs a type by joining each point of A to each point of B via a path. Topologically, it corresponds to the join of spaces, forming a space that interpolates between A and B.

Definition 147. (Formation). For types $A, B : \mathcal{U}$, the join $A * B : \mathcal{U}$.

Definition 148. (Constructors). The join is generated by the following higher inductive compositional structure:

$$A \vee B := \begin{cases} \mathrm{joinl} : A \to A \vee B \\ \mathrm{joinr} : B \to A \vee B \\ \mathrm{join}(a : A)(b : B) : \mathrm{joinl}(a) \equiv \mathrm{joinr}(b) \end{cases}$$

```
def ∨ (A : U) (B : U) : U
 := inductive { joinl (a: A)
              | joinr (b: B)
              | join (a: A) (b: B) : joinl(a) ≡ joinr(b)
              }
```

Theorem 39. (Elimination). For a type $C : \mathcal{U}$, maps $f : A \to C$, $g : B \to C$, and a family of paths $h : (a : A) \to (b : B) \to f(a) \equiv g(b)$, there exists a map $\mathrm{Ind}_\vee : A \vee B \to C$, such that:

$$\begin{cases} \mathrm{Ind}_\vee(\mathrm{joinl}(a)) = f(a) \\ \mathrm{Ind}_\vee(\mathrm{joinr}(b)) = g(b) \\ \mathrm{Ind}_\vee(\mathrm{join}(a, b, i)) = h(a, b, i) \end{cases}$$

```
def Ind∨ (A B C : U) (f : A -> C) (g : B -> C)
    (h : (a : A) -> (b : B) -> Path C (f a) (g b))
  : A ∨ B -> C
 := split { joinl a -> f a
          | joinr b -> g b
          | join a b @ i -> h a b @ i
          }
```

Theorem 40. (Computation). For all $z : A \vee B$, and predicate P, the rules of $\mathrm{Ind}_\vee$ hold for all parameters of the predicate P.

Theorem 41. (Uniqueness). Any two maps $h_1, h_2 : A \vee B \to C$ are homotopic if they agree on joinl, joinr, and join.

6.4.10 Colimit

Colimits construct the limit of a sequence of types, connected by maps, e.g., propositional truncations.

Definition 149. (Colimit) For a sequence of types $A : \mathrm{nat} \to \mathcal{U}$ and maps $f : (n : \mathbb{N}) \to An \to A(\mathrm{succ}(n))$, the colimit type $\mathrm{colimit}(A, f) : \mathcal{U}$.

$$\mathrm{colim} := \begin{cases} \mathrm{ix} : (n : \mathrm{nat}) \to An \to \mathrm{colimit}(A, f) \\ \mathrm{gx} : (n : \mathrm{nat}) \to (a : A(n)) \to \mathrm{ix}(\mathrm{succ}(n), f(n, a)) \equiv \mathrm{ix}(n, a) \end{cases}$$

```
def colimit (A : nat -> U) (f : (n : nat) -> A n -> A (succ n)) : U
 := inductive { ix (n : nat) (x: A n)
              | gx (n : nat) (a: A n)
                <i> [ (i=0) -> ix (succ n) (f n a),
                      (i=1) -> ix n a ]
              }
```

Theorem 42. (Elimination colimit) For a type $P : \mathrm{colimit}\, Af \to \mathcal{U}$, with $p : (n : \mathrm{nat}) \to (x : An) \to P(\mathrm{ix}(n, x))$ and $q : (n : \mathrm{nat}) \to (a : An) \to \mathrm{PathP}(\langle i \rangle P(\mathrm{gx}(n, a)@i))(p(\mathrm{succ}\, n)(fna))(pna)$, there exists $i : \prod_{x:\mathrm{colimit}\ Af} P(x)$, such that $i(\mathrm{ix}(n, x)) = pnx$.

6.4.11 Coequalizers

Coequalizer

The coequalizer of two maps $f, g : A \to B$ is a higher inductive type (HIT) that constructs a type consisting of elements in B, where f and g agree, along with paths ensuring this equality. It is a fundamental construction in homotopy theory, capturing the subspace of B where $f(a) = g(a)$ for $a : A$.

Definition 150. (Formation). For types $A, B : \mathcal{U}$ and maps $f, g : A \to B$, the coequalizer coeq $ABfg : \mathcal{U}$.

Definition 151. (Constructors). The coequalizer is generated by the following higher inductive compositional structure:

$$\text{Coeq} := \begin{cases} \text{inC} : B \to \text{Coeq}(A, B, f, g) \\ \text{glueC} : (a : A) \to \text{inC}(f(a)) \equiv \text{inC}(g(a)) \end{cases}$$

```
def Coeq (A B: U) (f g: A -> B) : U
 := inductive { inC (b: B)
              | glueC (a: A) : inC (f a) ≡ inC (g a)
              }
```

Theorem 43. (Elimination). For a type $C : \mathcal{U}$, map $h : B \to C$, and a family of paths $y : (x : A) \to \text{Path}_C(h(fx), h(gx))$, there exists a map coequRec : coeq $ABfg \to C$, such that:

$$\begin{cases} \text{coequRec}(\text{inC}(x)) = h(x) \\ \text{coequRec}(\text{glueC}(x, i)) = y(x, i) \end{cases}$$

```
def coequRec (A B C : U) (f g : A -> B) (h: B -> C)
    (y: (x : A) -> Path C (h (f x)) (h (g x)))
  : (z : coeq A B f g) -> C
 := split { inC x -> h x | glueC x @ i -> y x @ i }
```

Theorem 44. (Computation). For $z :$ coeq $ABfg$,

$$\begin{cases} \text{coequRec}(\text{inC } x) \equiv h(x) \\ \text{coequRec}(\text{glueC } x \mathbin{@} i) \equiv y(x) \mathbin{@} i \end{cases}$$

Theorem 45. (Uniqueness). Any two maps $h_1, h_2 :$ coeq $ABfg \to C$ are homotopic if they agree on inC and glueC, i.e., if $h_1(\text{inC } x) = h_2(\text{inC } x)$ for all $x : B$ and $h_1(\text{glueC } a) = h_2(\text{glueC } a)$ for all $a : A$.

Example 8. (Coequalizer as Subspace) The coequalizer coeq $ABfg$ represents the subspace of B, where $f(a) = g(a)$. For example, if $A = B = \mathbb{R}$ and $f(x) = x^2$, $g(x) = x$, the coequalizer captures the points where $x^2 = x$, i.e., $\{0, 1\}$.

Path Coequalizer

The path coequalizer is a higher inductive type that generalizes the coequalizer to handle pairs of paths in B. Given a map $\mathsf{p} : \mathsf{A} \to (\mathsf{b}_1, \mathsf{b}_2 : \mathsf{B}) \times (\mathrm{Path}_{\mathsf{B}}(\mathsf{b}_1, \mathsf{b}_2)) \times (\mathrm{Path}_{\mathsf{B}}(\mathsf{b}_1, \mathsf{b}_2))$, it constructs a type where elements of A generate pairs of paths between points in B, with paths connecting the endpoints of these paths.

Definition 152. (Formation). For types $\mathsf{A}, \mathsf{B} : \mathcal{U}$ and a map $\mathsf{p} : \mathsf{A} \to (\mathsf{b}_1, \mathsf{b}_2 : \mathsf{B}) \times (\mathsf{b}_1 \equiv \mathsf{b}_2) \times (\mathsf{b}_1 \equiv \mathsf{b}_2)$, there exists a path coequalizer $\mathrm{Coeq}_{\equiv}(\mathsf{A}, \mathsf{B}, \mathsf{p}) : \mathcal{U}$.

Definition 153. (Constructors). The path coequalizer is generated by the following higher inductive compositional structure:

$$\mathrm{Coequ}_{\equiv} := \begin{cases} \mathrm{inP} : \mathsf{B} \to \mathrm{Coeq}_{\equiv}(\mathsf{A}, \mathsf{B}, \mathsf{p}) \\ \mathrm{glueP} : (\mathsf{a} : \mathsf{A}) \to \mathrm{inP}(\mathsf{p}(\mathsf{a}).2.2.1@0) \equiv \mathrm{inP}(\mathsf{p}(\mathsf{a}).2.2.2@1) \end{cases}$$

```
data Coeq-≡ (A B: U) (p : A -> Σ (b1 b2: B), b1 ≡ b2 × b1 ≡ b2)
  = inP (b: B)
  | glueP (a:A) <i> [(i=0) -> inP ((p a).2.2.1 @ 0),
                     (i=1) -> inP ((p a).2.2.2 @ 1) ]
```

Theorem 46. (Elimination). For a type $\mathsf{C} : \mathcal{U}$, map $\mathsf{h} : \mathsf{B} \to \mathsf{C}$, and a family of paths $\mathsf{y} : (\mathsf{a} : \mathsf{A}) \to \mathsf{h}(\mathsf{p}(\mathsf{a}).2.2.1@0) \equiv \mathsf{h}(\mathsf{p}(\mathsf{a}).2.2.2@1)$, there exists a map $\text{Ind-Coequ}_{\equiv} : \mathrm{Coeq}_{\equiv}(\mathsf{A}, \mathsf{B}, \mathsf{p}) \to \mathsf{C}$, such that:

$$\begin{cases} \mathrm{coequPRec}(\mathrm{inP}(\mathsf{b})) = \mathsf{h}(\mathsf{b}) \\ \mathrm{coequPRec}(\mathrm{glueP}(\mathsf{a}, \mathsf{i})) = \mathsf{y}(\mathsf{a}, \mathsf{i}) \end{cases}$$

```
def Ind-Coequ≡ (A B C : U)
    (p : A -> Σ (b1 b2: B) (x: Path B b1 b2), Path B b1 b2)
    (h: B -> C) (y: (a : A) -> Path C (h (((p a).2.2.1) @ 0)) (h (((p a).2.2.2) @ 1)))
  : (z : coeqP A B p) -> C
 := split { inP b -> h b | glueP a @ i -> y a @ i }
```

Theorem 47. (Computation). For $\mathsf{z} : \mathrm{coeqP}\ \mathsf{A}\mathsf{B}\mathsf{p}$,

$$\begin{cases} \mathrm{coequPRec}(\mathrm{inP}\ \mathsf{b}) \equiv \mathsf{h}(\mathsf{b}) \\ \mathrm{coequPRec}(\mathrm{glueP}\ \mathsf{a} @ \mathsf{i}) \equiv \mathsf{y}(\mathsf{a}) @ \mathsf{i} \end{cases}$$

Theorem 48. (Uniqueness). Any two maps $\mathsf{h}_1, \mathsf{h}_2 : \mathrm{coeqP}\ \mathsf{A}\mathsf{B}\mathsf{p} \to \mathsf{C}$ are homotopic if they agree on inP and glueP, i.e., if $\mathsf{h}_1(\mathrm{inP}\ \mathsf{b}) = \mathsf{h}_2(\mathrm{inP}\ \mathsf{b})$ for all $\mathsf{b} : \mathsf{B}$ and $\mathsf{h}_1(\mathrm{glueP}\ \mathsf{a}) = \mathsf{h}_2(\mathrm{glueP}\ \mathsf{a})$ for all $\mathsf{a} : \mathsf{A}$.

6.4.12 K(G,n)

Eilenberg-MacLane spaces $\mathsf{K}(\mathsf{G},\mathsf{n})$ have a single non-trivial homotopy group $\pi_{\mathsf{n}}(\mathsf{K}(\mathsf{G},\mathsf{n})) = \mathsf{G}$. They are defined using truncations and suspensions.

Definition 154. (K(G,n)) For an abelian group G : abgroup, the type $\mathsf{KGn}(\mathsf{G})$: nat $\to \mathcal{U}$.

$$\mathsf{K}(\mathsf{G},\mathsf{n}) :\equiv \begin{cases} \mathsf{n} = 0 \rightsquigarrow \text{discreteTopology}(\mathsf{G}) \\ \mathsf{n} \geq 1 \rightsquigarrow \|\Sigma^{\mathsf{n}-1}(\mathsf{K1}'(\mathsf{G.1},\mathsf{G.2.1}))\|_{\mathsf{n}} \end{cases}$$

```
def KGn (G: abgroup) : N -> U
 := split { zero -> discreteTopology G
          | succ n -> nTrunc (Σ (K1' (G.1,G.2.1)) n) (succ n)
          }
```

Theorem 49. (Elimination KGn) For $\mathsf{n} \geq 1$, a type $\mathsf{B} : \mathcal{U}$ with isNGroupoid$(\mathsf{B}, \text{succ } \mathsf{n})$, and a map f : suspension$(\mathsf{K1}'\mathsf{G}) \to \mathsf{B}$, there exists $\text{rec}_{\mathsf{KGn}} : \mathsf{KGnG}(\text{succ } \mathsf{n}) \to \mathsf{B}$, defined via nTruncRec.

6.4.13 Localization

Localization constructs an F-local type from a type X, with respect to a family of maps $\mathsf{F_A} : \mathsf{S}(\mathfrak{a}) \to \mathsf{T}(\mathfrak{a})$.

Definition 155. (Localization Modality) For a family of maps $\mathsf{F_A} : \mathsf{S}(\mathfrak{a}) \to \mathsf{T}(\mathfrak{a})$, the F-localization $\mathsf{L}^{\mathsf{AST}}_{\mathsf{F}}(\mathsf{X}) : \mathcal{U}$.

$$\mathsf{L}^{\mathsf{A}}_{\mathsf{F}}(\mathsf{X}) := \begin{cases} \text{center} : \mathsf{X} \to \mathsf{L}_{\mathsf{F_A}}(\mathsf{X}) \\ \text{ext}(\mathfrak{a} : \mathsf{A}) \to (\mathsf{S}(\mathfrak{a}) \to \mathsf{L}_{\mathsf{F_A}}(\mathsf{X})) : \mathsf{T}(\mathfrak{a}) \to \mathsf{L}_{\mathsf{F_A}}(\mathsf{X}) \\ \text{isExt}(\mathfrak{a} : \mathsf{A})(\mathsf{f} : \mathsf{S}(\mathfrak{a}) \to \mathsf{L}_{\mathsf{F_A}}(\mathsf{X})) \to (\mathsf{s} : \mathsf{S}(\mathfrak{a})) : \text{ext}(\mathfrak{a}, \mathsf{f}, \mathsf{F}(\mathfrak{a}, \mathsf{s})) \equiv \mathsf{f}(\mathsf{s}) \\ \text{extEq}(\mathfrak{a} : \mathsf{A})(\mathfrak{g}, \mathfrak{h} : \mathsf{T}(\mathfrak{a}) \to \mathsf{L}_{\mathsf{F_A}}(\mathsf{X})) \\ \quad (\mathsf{p} : (\mathsf{s} : \mathsf{S}(\mathfrak{a})) \to \mathfrak{g}(\mathsf{F}(\mathfrak{a}, \mathsf{s})) \equiv \mathfrak{h}(\mathsf{F}(\mathfrak{a}, \mathsf{s}))) \\ \quad (\mathsf{t} : \mathsf{T}(\mathfrak{a})) : \mathfrak{g}(\mathsf{t}) \equiv \mathfrak{h}(\mathsf{t}) \\ \text{isExtEq} : (\mathfrak{a} : \mathsf{A})(\mathfrak{g}, \mathfrak{h} : \mathsf{T}(\mathfrak{a}) \to \mathsf{L}_{\mathsf{F_A}}(\mathsf{X})) \\ \quad (\mathsf{p} : (\mathsf{s} : \mathsf{S}(\mathfrak{a})) \to \mathfrak{g}(\mathsf{F}(\mathfrak{a}, \mathsf{s})) \equiv \mathfrak{h}(\mathsf{F}(\mathfrak{a}, \mathsf{s}))) \\ \quad (\mathsf{s} : \mathsf{S}(\mathfrak{a})) : \text{extEq}(\mathfrak{a}, \mathfrak{g}, \mathfrak{h}, \mathsf{p}, \mathsf{F}(\mathfrak{a}, \mathsf{s}) \equiv \mathsf{p}(\mathsf{s}) \end{cases}$$

```
data Localize (A X: U) (S T: A -> U) (F : (x:A) -> S x -> T x)
   = center (x: X)
   | ext (a: A) (f: S a -> Localize A X S T F) (t: T a)
   | isExt (a: A) (f: S a -> Localize A X S T F) (s: S a) <i>
     [ (i=0) -> ext a f (F a s) , (i=1) -> f s ]
   | extEq (a: A) (g h: T a -> Localize A X S T F)
     (p: (s : S a) -> Path (Localize A X S T F) (g (F a s)) (h (F a s)))
     (t : T a) <i> [ (i=0) -> g t , (i=1) -> h t ]
   | isExtEq (a: A) (g h : T a -> Localize A X S T F)
     (p: (s : S a) -> Path (T a -> Localize A X S T F) (g (F a s)) (h (F a s)))
     (s : S a) <i> [ (i=0) -> extEq a g h p (F a s) , (i=1) -> p s ]
```

Theorem 50. (Localization Induction) For any $\mathsf{P} : \Pi_{\mathsf{X}:\mathcal{U}}\mathsf{L}_{\mathsf{F_A}}(\mathsf{X}) \to \mathcal{U}$ with $\{\mathfrak{n}, \mathsf{r}, \mathsf{s}\}$, satisfying coherence conditions, there exists $\mathfrak{i} : \Pi_{\mathsf{x}:\mathsf{L}_{\mathsf{F_A}}(\mathsf{X})}\mathsf{P}(\mathsf{x})$, such that $\mathfrak{i} \cdot \text{center}_{\mathsf{X}} = \mathfrak{n}$.

Conclusion

HITs directly encode CW-complexes in HoTT, bridging topology and type theory. They enable the analysis and manipulation of homotopical types.

6.5 Висновки

Розділ 7

Додаткові матеріали

Присвячується майстрам
тибетського буддизму

Атіша, Нагарджуна, Лонгченпа

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

7.1 Формалізація мадх'яміки

Зараз я дам вам відчути смак формальної філософії по-справжньому! А то вам може здатися, що це канал з формальної математики, а не формальної філософії. Я ж вважаю, що якщо формальна філософія не спирається на формальну математику, то гріш ціна такій формальній філософії.

```
module buddhism where
import path
```

Сьогодні ми будемо формалізувати поняття недвоїстості в буддизмі, яке пов'язане одразу з багатьма концепціями на рівнях Сутри, Тантри та Дзогчена: поняттям взаємозалежного виникнення та поняттям порожнечі всіх феноменів (Сутра Праджняпараміти). Класичний приклад із розчленовуванням тіла ставить питання, коли тіло перестає бути людиною-істотою, якщо від нього почати відрубувати шматки м'яса (ми буддисти любимо і лілеєм такі уявні образи-експерименти) або іншими словами, щоб відрізнити тіло від не-тіла, нам потрібен двомісний предикат (родина типів), функція, яка може ідентифікувати конректні два еклемпляри тіла. Практично йдеться про ідентифікацію двох об'єктів, тобто про звичайний тип-рівность Мартіна-Льофа.

За фреймворк візьмемо концепти Готтлоба Фреге, згідно з визначенням, концепт - це предикат над об'єктом або, іншими словами, Пі-тип Мартіна-Льофа, індексований тип, сім'я типів, тривіальне розшарування тощо. Де об'єкт x з o належить концепту, якщо сам концепт, параметризований цим об'єктом, населений p(o) : U (де p : concept o).

```
concept (o: U): U
  = o -> U
```

Концепт p повинен надавати приклад чи контрприклад розрізнення, тобто щоб визначити тіло це чи не тіло ще, поки ми його розчленовуємо, нам потрібно як мінімум два шматки: тіло і не тіло як приклади ідентифікації. Таким чином, недвоїстість може бути представлена як рівність між усіма розшаруваннями (предекатами над об'єктами).

```
nondual (o: U) (p: concept o): U
  = (x y: o) -> Path U (p x) (p y)
```

Отже, недвоїстість усуває різницю між прикладами і контрприкладами на примордіальному рівні мандали MLTT, тобто ідентифікує всі концепти. Сама ж ідентифікація класів об'єктів, які належать різним концептам — це умова, що стискає всі об'єкти в точку, або стягуваний простір, вершина конуса мандали MLTT, або, іншими словами, порожнеча всіх феноменів виражена як тип логічної одиниці, який містить лише один елемент.

```
allpaths (o: U): U
  = (x y: o) -> Path o x y
```

Формулювання буддійської теореми недвоїстості, яка поширюється всі типи учнів (тупих, середніх і тямущих), може звучати так: недвоїстість концепту є спосіб ідентифікації його об'єктів. Сформулюємо цю саму теорему в інший бік: спосіб ідентифікації об'єктів задає предикат неподвоїстості концептів. Туди — ((p: concept o) -> nondual o p) -> allpaths o, Сюди — allpaths o -> ((p: concept o) -> nondual o p). І доведемо її! Як видно з сигнатур нам лише треба побудувати функцію транспорту між двома просторами шляхів: (p x) $=_U$ (p y) і x $=_o$ y. Скористаємося приведенням шляху до стрілки (coerce) та конгруентності (cong) з базової бібліотеки.

```
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)
```

Як бачите, теоремка про порожнечу всіх феноменів вийшла на кілька рядків, які демонструють: 1) основи формальної філософії та швидке занурення в область математичної філософії; 2) гарний приклад до першого розділу HoTT на простір шляхів та модуль path[1].

[1] `https://favonia.org/files/thesis.pdf`

7.2 Формалізація вводу-виводу для Coq/O-Caml

```
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} (s: nat) (x: Co effect A) : Lwt.t A :=
  match s with
    | O => error tt
      | S s =>
    match x with
      | Bind _ _ x f => Lwt.bind   (eval_aux s x) (fun v_x => eval_aux s (f v_x))
      | Split _ x y  => Lwt.choose (eval_aux s x) (eval_aux s y)
      | Join _ _ x y => Lwt.join   (eval_aux s x) (eval_aux s 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.
```

7.3 Формалізація вводу-виводу для PTS/BEAM

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)
```

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
```

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>
```

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>
```

7.4 Словник термінів

З області програмування:

Формальні методи —
Система типізації —
Типова сигнатура —
Імплементація —
Інтерпретатор —
Мова програмування —
Теорія типів —
Компіляція —
Базова бібліотека —
Середовище виконання —
Вищі мови програмування —
BNF нотація —
Синтаксичне дерево —
Синтаксис —
Семантика —

З обсласті математики:

STLC —
Логіка першого порядку —
Класичні логіки —
Лямбда-числення —
Лямбда-куб —
Пі-числення —
Модальні логіки —
Основи математики —
Математична (формальна) верифікація —
Теорія категорій —
Теорія топосів —
Теорема Гьоделя про неповноту —
Логіки вищий порядків —
Ізоморфізм —
Гомотопічна теорія типів —
Диференціальна топологія —
Диференціальна геометрія —
Еквіваріантна модальна супер-гомотопічна система —
Теорія залежних типів —
Числення конструкцій —
Декартово-замкнена категорія —
Числення індуктивних конструкцій —
Системи доведення теорем (прувери) —
Фібраційні математичні прувери —
PTS система —
MLTT система —

Бібліографія

[1] 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.

[2] T. Streicher, *Semantics of type theory - correctness, completeness and independence results*. Progress in theoretical computer science, Birkhäuser, 1991.

[3] P. Dybjer, "Internal type theory," in *International Workshop on Types for Proofs and Programs*, pp. 120–134, Springer, 1995.

[4] M. Hofmann, *Syntax and semantics of dependent types*. London: Springer London, 1997.

[5] S. Awodey, "Natural models of homotopy type theory," 2017.

[6] P. L. Lumsdaine and M. A. Warren, "The local universes model," *ACM Transactions on Computational Logic*, vol. 16, p. 1–31, Jul 2015.

[7] A. Grothendieck, *Revêtements étales et groupe fondamental (SGA 1)*, vol. 224 of *Lecture notes in mathematics*. Springer-Verlag, 1971.

[8] J. Cartmell, "Generalised algebraic theories and contextual categories," *Annals of Pure and Applied Logic*, vol. 32, pp. 209–243, 1986.

[9] T. Ehrhard, "A categorical semantics of constructions," in *Proceedings Third Annual Symposium on Logic in Computer Science*, (Los Alamitos, CA, USA), pp. 264–273, IEEE Computer Society, 1988.

[10] B. Jacobs, "Comprehension categories and the semantics of type dependency," *Theoretical Computer Science*, vol. 107, no. 2, pp. 169–207, 1993.

[11] B. Jacobs, *Categorical logic and type theory*, vol. 141. Elsevier, 1999.

[12] D. G. Quillen, *Homotopical algebra*. Springer, 1967.

[13] T. Coquand, B. Mannaa, and F. Ruch, "Stack semantics of type theory," 2017.

[14] 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.

[15] 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.

[16] N. G. de Bruijn, *AUTOMATH, a Language for Mathematics*, pp. 159–200. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983.

[17] P. Martin-Löf and G. Sambin, *Intuitionistic type theory*. Studies in proof theory, Bibliopolis, 1984.

[18] T. Coquand and G. Huet, "The calculus of constructions," in *Information and Computation*, (Duluth, MN, USA), pp. 95–120, Academic Press, Inc., 1988.

[19] 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.

[20] 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.

[21] S. P. Jones and E. Meijer, "Henk: A typed intermediate language," in *In Proc. First Int'l Workshop on Types in Compilation*, 1997.

[22] C.-E. Ore, "The extended calculus of constructions (ecc) with inductive types," in *Information and Computation*, vol. 99, pp. 231 – 264, 1992.

[23] 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.

[24] 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.

[25] A. Stump, "The calculus of dependent lambda eliminations," in *Journal of Functional Programming*, vol. 27, Cambridge University Press, 2017.

[26] G. Barthe, *Extensions of pure type systems*, pp. 16–31. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995.

[27] 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.

[28] H. Geuvers, *Induction Is Not Derivable in Second Order Dependent Type Theory*, pp. 166–181. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[29] S. Awodey, "Impredicative encodings in hott," 2017.

[30] S. Speight, "Impredicative encoding of inductive types in hott," 2017.