Дисертація

Концептуальна модель
системи доведення теорем
на основі гомотопічної теорії типів

Максим Сохацький

Зміст

ВСТУП

Присвячується Маші та Міші

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

Актуальність роботи

Ціна помилок в індустрії надзвичайно велика. Наведемо відомі приклади: 1) Mars Climate Orbiter (1998), помилка невідповідності типів британської метричної системи, коштувала 80 мільйонів фунтів стерлінгів. Невдача стала причиною переходу NASA повністю на метричну систему в 2007 році. 2) Ariane Rocket (1996), причинан катастрофи – округлення 64-бітного дійсного числа до 16-бітного. Втрачені кошти на побудову ракети та запуск 500 мільйонів 3) Помилка в FPU в перших Pentium (1994), збитки на 300 мільйонів. 4) Помилка в SSL (heartbleed), оцінені збитки у розмірі 400 мільйонів. 5) Помилка у логіці бізнес-контрактів EVM та DAO (неконтрольована рекурсія), збитки 50 мільйонів. Більше того, і найголовніше, помилки у програмному забезпеченні можуть коштувати життя людей.

Формалізована постановка задачі

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

Об'єкт та предмет дослідження

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

Предметом дослідження такої системи мов є теорія типів, яка вивчає обчислювальні властивості мов. Теорія типів виділилася в окрему науку Пером Мартіном-Льофом як запит на вакантне місце у трикутнику теорій, які відповідають ізоморфізму Каррі-Говарда-Ламбека (Логіки, Мови, Категорії). Інші дві це: теорія категорій та логіка вищих порядків.

Мета і задачі дослідження

Одна з причина 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.

Формалізація семантики відбувається завдяки теорії категорій, яка є абстрактною алгеброю фунцій, метематичним інструментом для формалізації мов програмування та довільних математичних теорій які описуються логіками вищих порядків.

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

Методи дослідження

Незалежно від піходу до верифіції, формальна верифікація неможлива, якщо мова програмування моделі формально не визначена. Це означає шо значна міра програмного забезпечення може бути автоматично верифікована тільки для тих мов, формальні моделі яких побудовані, на даний момент це тільки мова C. Більше того, не завжди можна також формально довести те, що програма завершиться, потрібно звужувати клас програм, якщо формальні специфікації містять такі властивості.

Глава 1

ФОРМАЛЬНА ВЕРИФІКАЦІЯ

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

1.1 Формальна верифікація та валідація

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

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

[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

дель, 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, та 2) Модельно-орієнотований підхід, де модель створена конструктивними побудовами, як то на базі теорії множин, чи інкаше, а системні операції визначаються тим, як вони змінюють стан системи. Також були створені сімейства послідованих та розподілених мов.

1.2.1 Протоколи

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

1.2.2 Бізнес-процеси

Інша відома мова формальної специфікації як стандарт для моделювання розподілених систем, таких як телефонні мережі та протоколи, це LOTOS[4] (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+[5], CSP (Hoare, 1985), CCS[6] (Milner, 1971), Actor Model, Reactive Streams, BPMN, etc.

1.2.3 Теореми

1.3 Формальні методи верифікації

1.3.1 Спеціалізовані системи моделювання

Можна виділити три підходи до верифікації. Перший застосовується де вже є певна програма написана на певній мові програмування і потрібно довести ізоморфність цієї програми до доведеної моделі. Ця задача вирішується у побудові теоретичної моделі для певної мови програмування, потім програма на цій

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

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

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

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

мові переводиться у цю теоретичну модель і доводить ізоморфізм цієї програми у побудованій моделі до доведеної моделі. Приклади таких систем та піходів: 1) VST (CompCert, сертифікація C програм), 2) NuPRL (Cornell University, розподілені системи, залежні типи), 3) TLA+ (Microsoft Reseach, Леслі Лампорт), 4) Twelf (для верифікації мов програмування), 5) SystemVerilog (для ч'програмного та апаратного забезпечення).

1.3.2 Мови з залежними типами

Другий підхід можна назвати підходом вбудованих мов. Компілятор основої мови перевіряє модель закодовану у ній же. Можливо моделювання логік вищого порядку, лінійних логік, модальних логік, категорний та гомотопічних логік. Процес специфікації та верифікації відбувається в основній мові, а сертифіковані програми автоматично екстрагуються в довільні мови. Приклади таких систем: 1) Coq побудована на мові OCaml від науково-дослідного інституту Франції INRIA; 2) Agda побудовані на мові Haskell від шведського інституту технологій Чалмерс; 3) Lean побудована на мові C++ від Microsoft Research та Університету Каргені-Мелона; 4) Idris подудована на мові Haskell Едвіна Бреді з шотландського Університету ім. св. Андрія; 5) F* – окремий проект Microsoft Research.

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

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

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

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

1.4 Формальні середовища виконання

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

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

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

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

1.4.1 Інтерпретатори на Rust

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

1.4.2 Коіндуктивні системи на Coq/OCaml

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 на Erlang

Третій прототип – побудова тайпчекера та ектрактора у мову Erlang та O. Ця робота представлена у вигляді PTS тайпчекера OM, 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.

1.4.4 Гомотопічні системи на Haskell

Глава 2

ГЕНЕЗА

Другий розділ описує розвиток концептуальної моделі системи доведення теорем у розрізі різних семантичних рівнів системи, показуючи її еволюцію з підвищенням складності мов. Так, виділяється наступна послідовність мов, та функторів між ними, де кожна мова-кодомен є складнішою та біль потужною за мову-домен. Перша мова являє собою нетипизоване лямбда числення, остання мова містить обчислювальну семантику гомотопічної теорії типів.

$$\mathrm{O}_{\mathrm{CPS}} \to \mathrm{O}_{\mathrm{PTS}} \to \mathrm{O}_{\mathrm{MLTT}} \to \mathrm{O}_{\mathrm{CiC}} \to \mathrm{O}_{\mathrm{CCHM}}$$

Кожна мова програмування може бути доменом або кодоменом морфізмів в категорії мов програмування. Тут представлена спектральна категорія, об'єкти якої можна пронумерувати в залежності від позиції в генетичній послідовності.

Мови програмування. Мова програмування — це індуктивний тип конструкторів мови, для якої існує операційна семантка (правила обчислень) та правила виводу. Найпростіша мова програмування — нетипизоване лямбда числення, ізоморфне екстракту в Erlang.

$$\mathit{type}, \mathit{erase}, \mathit{norm}, \mathit{opt} : \mathrm{O}_{\mathrm{PTS}} \to \mathrm{O}_{\mathrm{PTS}}$$

$$\mathit{extract} : \mathrm{O}_{\mathrm{PTS}} \to \mathrm{O}_{\mathrm{CPS}}$$

$$\mathit{certify} : \mathrm{O}_{\mathrm{PTS}} \to \mathrm{O}_{\mathrm{CPS}} = \mathit{type} \circ \mathit{norm} \circ \mathit{opt} \circ \mathit{erase} \circ \mathit{extract}$$

Об'єкти. Об'єкти категорій — мови програмування. Кожна мова програмування анонсує систему типів згідно свого індуктивного синтаксичного дерева. Усі можливі екземпляри цього синтаксичного дерева є усіма можливими программами в цій мові програмування.

Мовні Категорії. Мовна категорія — це категорія, єдиний об'єкт якої це синтаксичне дерево мови, а морфізми — це стрілки цієї maybe-категорії: [norm,type,infer,erase,extract]. Стрілки зокрема містять правила виводу, типизації, нормалізації, екстактів, тощо.

Функторіальні мовні перетворення: 1) extract: maybe A -> maybe B — з однієї мови програмування A в іншу мову програмування B; 2) type: maybe A -> maybe A — перевірка терма первної мови програмування; 3) infer: maybe A -> maybe A — типізація; 4) norm: maybe A -> maybe A — нормалізація.

Вхідні синтаксиси. Специфікації Увесь спектр мов програмування, що сприймаються системою визначається набором синтаксисів, парсери яких на виході дають індуктивні синтаксичні дерева (закодовані у Бом, IR/II, чи будь-якому довільному індуктивному кодуванню).

Вихідні синтаксиси. Інтерпретатори

2.1 Інтерпретатор

Кількість мов прототипа обмежена двома інтерпретаторами: O та Erlang, однак система не обмежується цими мовами, а має експериментальне HM ядро з екстрактом в C++. Цікаво було би отримати екстракт в Rust.

```
data O
   = var (n: nat)
   | app (f a: O)
   | lambda (x: nat) (d c: O)
```

Вхідними синтаксисами екстраторів є синтаксиси відповідної мови ядра. На даний момент в роботі ми підтримуємо PTS та індуктивний синтаксиси.

The motivation for building an interpreter that can run on L1 cache sizes (limited to 64KB of code and data) is based on success of LuaJIT, V8, HotSpot, and vector lanaguages such as K and J. If we can build a really fast VM for such interpreter (compact aligned bytecode) and make it stick to L1 sizes with enabled AVX instruction set then it can outperform any reasonable alternative.

This approach was chosen for building O-CPS interpreter in Rust. Following results were achieved in factorial (5) and ackerman function (3,4) benchmarks, and that's even without bytecode VM implementation! Next step to improvement.

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


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

The key challenge here was that O-CPS implementation was made in Rust's flavour of linear types. Thus lambda interpreter as an internal language of cartesian closed category was written in a language with linear types and lifetimes as an internal language of symmetric monoidal categories. Such an exercise will definitely increase your confidence in linear types.

2.1.1 Векторизація засобами мови Rust

```
(&AST::Atom(Atom::Value(Value::$vec(ref l))),
 &AST::Atom(Atom::Value(Value::$vec(ref r)))) => {
    let a: Vec<$atype> = l.iter()
        .zip(r)
        .map(|(l,r)| dyad_map_expr!(l,$op,r,$atom))
        .collect::>();
    Ok( AST::Atom(Atom::Value(Value::$r_vec(a))) )
},
```

```
objdump ./target/release/o -d | grep mulpd
   223f1: c5 f5 59 0c d3    vmulpd (%rbx,%rdx,8),%ymm1,%ymm1
   223f6: c5 dd 59 64 d3 20 vmulpd 0x20(%rbx,%rdx,8),%ymm4,%ymm4
   22416: c5 f5 59 4c d3 40 vmulpd 0x40(%rbx,%rdx,8),%ymm1,%ymm1
   2241c: c5 dd 59 64 d3 60 vmulpd 0x60(%rbx,%rdx,8),%ymm4,%ymm4
   2264d: c5 f5 59 0c d3    vmulpd (%rbx,%rdx,8),%ymm1,%ymm1
   22652: c5 e5 59 5c d3 20 vmulpd 0x20(%rbx,%rdx,8),%ymm3,%ymm3
```

2.1.2 Синтаксис

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

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


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

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

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

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

lazy n (Assign n b)               e k = D n b (ContAssign n k)
     n (Cond v a l)               e k = D n v (ContCond l r k)
     n (List l)                   e k = eve n l e k
     n (Call c a)                 e k = D n a (ContCall c k)
     n (Name s)                   e k = cont (lookup n s e) k
     n (Lambda _ x y)             e k = cont ((Lambda n x y) n) k

evf  n (Lambda c x y) a e k = cont (y (if (= c []) n c)) (ContFunc x a k)
     n       (Name s) a e k = lookup n s e
eve  n       (Cons a d) e k = D n a (ContExpr d k)
     n              Nil e k = cont n Nil e k
     n                a e k = D n a k
emr            (Dict x) e k = R Dict (rev x)
               (Cons x y) e k = R Dict (Cons (rev x) (rev x))
                        a e k = R a
run  D              n a e k = lazy n a e k
     R                a     = a

cont n (Dict v) e (ContCall c k)          = evf n c v e k
     n x        e (ContCall c k)          = evf n c x e k
     n x        e (ContFunc s a k)        = eve (e.define_args s (rev a)) v k
     n False    e (ContCond l r k)        = D n r k
     n True     e (ContCond l r k)        = D n l k
     n x        e (ContCond l r k)        = D n x (ContCond l r k)
     n x        e (ContAssign (Name s) k) = eve n (e.define s x) k
     n x        e (ContExpr (Cons a d) k) = eve n (Cons a d) k
     n x        e (ContExpr r k)          = cont n x k
     n x        e  ContRet                = emr x e k
```

2.2 Чиста система

2.3 Introduction

IEEE[1] standard and ESA[2] regulatory documents define a number of tools and approaches for verification and validation processes. The most advanced techniques involve mathematical languages and notations. The age of verified math was started by de Bruin's AUTOMATH prover and Martin-Löf [?]'s type theory. Today we have Coq, Agda, Lean, Idris, F* languages which are based on Calculus of Inductive Constructions or CiC [?]. The core of CiC is Calculus of Constructions or CoC [?]. Further development has lead to Lambda Cube [?] and Pure Type Systems by Henk [?] and Morte[3]. Pure Type Systems are custom languages based on CoC with single Pi-type and possibly other extensions. Notable extensions are ECC, ECC with Inductive Types [?], K-rules [?]. The main motivation of Pure Type Systems is an easy reasoning about core, strong normalization and trusted external verification due to compact type checkers. A custom type checker can be implemented to run certified programs retrieved over untrusted channels. The applications of such minimal cores are 1) Blockchain smart-contract languages, 2) certified applications kernels, 3) payment processing, etc.

2.3.1 Generating Trusted Programs

According to Curry-Howard, a correspondence inside Martin-Löf Type Theory [?] proofs or certificates are lambda terms of particular types or specifications. As both specifications and implementations are done in a typed language with dependent types we can extract target implementation of a certified program just in any programming language. These languages could be so primitive as untyped lambda calculus and are usually implemented as untyped interpreters (JavaScript, Erlang, PyPy, LuaJIT, K). The most advanced approach is code generation to higher-level languages such as C++ and Rust (which is already language with trusted features on memory, variable accessing, linear types, etc.). In this work, we present a simple code extraction to Erlang programming language as a target interpreter. However, we have also worked on C++ and Rust targets as well.

[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] Gabriel Gonzalez. Haskell Morte Library

2.3.2 System Architecture

Om as a programming language has a core type system, the PTS$^\infty$ — the pure type system with the infinite number of universes. This type system represents the core of the language. Higher languages form a set of front-ends to this core. Here is example of possible languages: 1) Language for inductive reasoning, based on CiC with extensions; 2) Homotopy Core with interval [0,1] for proving J and funExt; 3) Stream Calculus for deep stream fusion (Futhark); 3) Pi-calculus for linear types, coinductive reasoning and runtime modeling (Erlang, Ling, Rust). These languages desugar to PTS$^\infty$ as an intermediate language before extracting to target language[4].

Not all terms from higher languages could be desugared to PTS. As was shown by Geuvers [?] we cannot build induction principle inside PTS, we need a fixpoint extension to PTS. And also we cannot build the J and funExt terms. But still PTS is very powerful, it's compatible with System F libraries. The properties of that libraries could be proven in higher languages with Induction and/or [0,1] Homotopy Core. Then runtime part could be refined to PTS, extracted to target and run in an environment.

We see two levels of extensions to PTS core: 1) Inductive Types support; 2) Homotopy Core with [0,1] and its eliminators. We will touch a bit this topic in the last section of this document.

PTS синтаксиси. Мінімальне ядро з однією аксіомою сприймає декілька лямбда ситаксисів. Перший синтаксис сумісний з системою програмування $\mathbf{morte}$[5], та походить від неї. Інший синтаксис сумісний з синтаксисом $\mathbf{cubical}$[6]. Планувалося також підтримати синтаксис $\mathbf{caramel}$[7].

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

Усі терми підчиняються системі аксіом $\mathbf{Axioms}$ всередині послідовності всесвітів $\mathbf{Sorts}$ та складність залежного терму відповідає

[4]Note that extracting from [0,1] Homotopy Core is an open problem
[5]http://github.com/Gabriel439/Haskell-Morte-Library
[6]http://github.com/mortberg/cubicaltt
[7]https://github.com/MaiaVictor/caramel

Таблица 2.1 : List of languages, tried as verification targets

Target	Class	Intermediate	Theory
C++	compiler/native	HNC	System F
Rust	compiler/native	HNC	System F
JVM	interpreter/native	Java	F-sub[8]
JVM	interpreter/native	Scala	System F-omega
GHC Core	compiler/native	Haskell	System D
GHC Core	compiler/native	Morte	CoC
Haskell	compiler/native	Coq	CiC
OCaml	compiler/native	Coq	CiC
BEAM	interpreter	Om	PTS$^\infty$
O	interpreter	Om	PTS$^\infty$
K	interpreter	Q	Applicative
PyPy	interpreter/native	N/A	ULC
LuaJIT	interpreter/native	N/A	ULC
JavaScript	interpreter/native	PureScript	System F

максимальній складності домена та кодомена (правила $\mathfrak{Rules}$). Таким чином визначається простір всесвітів, та його конфігурація може бути записана згідно нотації Барендрехта для систем з чистими типами:

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

An intermediate Om language is based on Henk [6] languages described first by Erik Meijer and Simon Peyton Jones in 1997. Leter on in 2015 Morte impementation of Henk design appeared in Haskell, using Boem-Berrarducci encoding of non-recursive lambda terms. It is based only on one type constructor Π, its special case λ and theirs eliminators: $\mathfrak{apply}$ and $\mathfrak{curry}$, infinity number of universes, and one computation rule called β-reduction. The design of Om language resemble Henk and Morte both design and implementation. This language indended to be small, concise, easy provable and able to produce verifiable peace of code that can be distributed over the networks, compiled at target with safe trusted linkage.

2.3.3 Синтаксис

Om syntax is compatible with $\lambda\mathbf{C}$ Coquand's Calculus of Constructions presented in Morte and Henk languages. However it has extension in a part of specifying universe index as a Nat number.

```
<> ::= #option
 I ::= #identifier
 U ::= * < #number >
 O ::= U
     | I | ( O ) | O O | O → O
     | λ ( I : O ) → O
     | ∀ ( I : O ) → O
```

Equivalent tree encoding for parsed terms is following:

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

2.3.4 Всесвіти

The OM language is a higher-order dependently typed lambda calculus, an extension of Coquand's Calculus of Constructions with the predicative/impredicative hierarchy of indexed universes. This extension is motivated avoiding paradoxes in dependent theory. Also there is no fixpoint axiom needed for the definition of infinity term dependance.

```
U₀ : U₁ : U₂ : U₃ : ...

U₀ --- propositions
U₁ --- values and sets
U₂ --- types
U₃ --- sorts
```

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

Предикативні всесвіти

All terms obey the A ranking inside the sequence of S universes, and the complexity R of the dependent term is equal to a maximum of the term's complexity and its dependency. The universes system is completely described by the following PTS notation (due to Barendregt):

Note that predicative universes are incompatible with Church lambda term encoding. You can switch predicative vs impredicative uninverses by typecheker parameter.

$$(A_1) \qquad \frac{i : Nat, j : Nat, i < j}{U_i : U_j}$$

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

Імпредикативні всесвіти

Propositional contractible bottom space is the only available extension to predicative hierarchy that not leads to inconsistency. However there is another option to have infinite impredicative hierarchy.

$$(A_2) \qquad \frac{i : Nat}{U_i : U_{i+1}}$$

$$(R_2) \qquad \frac{i : Nat, \quad j : Nat}{U_i \to U_j : U_j}$$

2.3.5 Контексти

The contexts model a dictionary with variables for type checker. It can be typed as the list of pairs or List Sigma. The elimination rule is not given here as in our implementation the whole dictionary is destroyed after type checking.

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

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

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

2.3.6 Операційна семантика

This language is called one axiom language (or pure) as eliminator and introduction rules inferred from type formation rule. The only computation rule of Pi type is called beta-reduction. Computational rules of language are called operational semantics and establish equality of substitution and lambda application. Operational semantics in that way defines the rewrite rules of computations.

$$\frac{A : \mathcal{U}_i \ , \ x : A \vdash B : \mathcal{U}_j}{\Pi\,(x : A) \to B : \mathcal{U}_{p(i,j)}} \qquad (\Pi\text{-formation})$$

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

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

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

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

The theorems (specification) of PTS could be embedded in itself and used as Logical Framework for the Pi type. Here is the example in the higher language.

```
PTS (A: U): U
  = (Pi_Former: (A -> U) -> U)
  * (Pi_Intro: (B: A -> U) (a: A) -> B a -> (A -> B a))
  * (Pi_Elim: (B: A -> U) (a: A) -> (A -> B a) -> B a)
  * (Pi_Comp1: (B: A -> U) (a: A) (f: A -> B a) ->
    Path (B a) (Pi_Elim B a (Pi_Intro B a (f a))) (f a))
  * (Pi_Comp2: (B: A -> U) (a: A) (f: A -> B a) ->
    Path (A -> B a) f (\(x:A) -> f x))
```

The proofs intentionally left blank, as it proofs could be taken from various sources [?]. The equalities of computational semantics presented here as Path types in the higher language. The Om language is the extention of the PTS$^\infty$ with the remote AST node which means remote file loading from trusted storage, anyway this will be checked by the type checker. We deny recursion over the remote node. We also add an index to var for simplified de Bruijn indexes, we allow overlapped names with tags, incremented on each new occurrence. Our typechecker differs from cannonical example of Coquand [?]. We based our typechecker on variable Substitution, variable Shifting, term Normalization, definitional Equality anf Type Checker itself.

2.3.7 Перевірка типів

For sure in a pure system, we should be careful with :remote AST node. Remote AST nodes like #List/Cons or #List/map are remote links to files. So using trick one should desire circular dependency over :remote.

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

2.3.8 Індекси де Брейна

Shift renames var N in B. Renaming means adding 1 to the nat component of variable.

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

2.3.9 Підстановка, нормалізація, рівність

Substitution replaces variable occurance in terms.

```
 sub (:star,X)      N V L → (:star,X)
     (:var,N,L)     N V L → V
     (:var,N,I)     N V L → (:var,N,I-1) when I > L
     (:remote,X)    N V L → (:remote,X)
     (:pi,N,O,I,O) N V L → (:pi,N,O,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,O,I,O) N V L → (:fn,N,O,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)
```

Normalization performs substitutions on applications to functions (beta-reduction) by recursive entrance over the lambda and pi nodes.

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

Definitional Equality simply checks the equality of Erlang terms.

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

2.3.10 Використання мови

Here we will show some examples of Om language usage. In this section, we will show two examples. One is lifting PTS system to MLTT system by defining Sigma and Equ types using only Pi type. We will use Bohm inductive dependent encoding [?]. The second is to show how to write real world programs in Om that performs input/output operations within Erlang environment. We show both recursive (finite, routine) and corecursive (infinite, coroutine, process) effects.

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

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

2.3.11 Екстракти

This works expect to compile to limited target platforms. For now Erlang, Haskell and LLVM is awaiting. Erlang version is expected to be useful both on LING and BEAM Erlang virtual machines.

2.3.11.1 Інтерпретатори

From a practical point of view, we develop Erlang with Dependent Types. Thus we carefully integrate with Erlang platform by generating Erlang AST and trying to be compatible with Erlang kernel through mapping of inductive types to underlying Erlang primitives such as process, receive, and spawn. Erlang extraction is sponsored and supported by Synrc Research Center. Extraction in other languages could also be easily implemented.

2.3.11.2 LLVM

Another branch of research is dedicated to evaluation of LLVM lambda generation. It could be direct MIR or LLVM generation, or we could generate Rust/C++ code that could be passed to LLVM optimizer. If you are interested in LLVM target, please take a look at github.com/nponeccop/HNC.

2.3.11.3 FPGA

We are very interested in compilation to FPGA. As was shown with interaction nets it is possible to compact packaging of inductive construction in silicon, giving back the inner language of space to the natural encoding. If you are interested in moving this project forward and have a vision how to do it please drop us a line in gitter.im/groupoid/exe chat.

2.4 Індуктивна система

Індуктивні синтаксиси. 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.

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

```
data tele   (A: U) = nil | tel (n: name) (b: A) (xs: tele A)
data branch (A: U) =        br (n: name) (a: list name) (t: A)
data label  (A: U) =       lab (n: name) (t: tele A)

data Inductive (A: U)
   = parent (p: PTS (Inductive A))
   | sum  (n: name) (t: tele A) (labels:   list (label A))
   | case (n: name) (t: ind)    (branches: list (branch A))
   | ctor (n: name)             (args:     list A)
```

2.4.1 Поліноміальні функтори

There are two types of recursion: one is least fixed point (as $F_A\ X = 1 + A \times X$ or $F_A\ X = A + X \times X$), in other words the recursion with a base (terminated with a bounded value), lists and trees are examples of such recursive structures (so we call induction recursive sums); and the second is greatest fixed point or recursion withour a base (as $F_A\ X = A \times X$) — such kind of recursion on infinite lists (codata, streams, coinductive types) we can call recursive products. Least fixed point trees are called well-founded trees and encode polynomial functors.

Natural Numbers: $\mu\ X \to 1 + X$
List A: $\mu\ X \to 1 + A \times X$
Lambda calculus: $\mu\ X \to 1 + X \times X + X$
Stream: $\nu\ X \to A \times X$
Potentialy Infinite List A: $\nu\ X \to 1 + A \times X$
Finite Tree: $\mu\ X \to \mu\ Y \to 1 + X \times Y = \mu\ X = \mathrm{List}\ X$

As we know there are several ways to appear for variable in recursive algebraic type. Least fixpoint are known as an recursive expressions that have a base of recursion Both recursive and corecursive datatypes could be encoded using Boem-Berarducci encoding as an non-recursive definitions of folds that include in indentity signature all the constructor components of (co)inductive type.

2.4.2 Кодування Бома

The data type of lists over a given set A can be represented as the initial algebra $(\mu L_A, \mathbf{in})$ of the functor $L_A(X) = 1 + (A \times X)$. Denote $\mu L_A = \mathbf{List}(A)$. The constructor functions $\mathbf{nil} : 1 \to \mathbf{List}(A)$ and $\mathbf{cons} : A \times \mathbf{List}(A) \to \mathbf{List}(A)$ are defined by $\mathbf{nil} = \mathbf{in} \circ \mathbf{inl}$ and $\mathbf{cons} = \mathbf{in} \circ \mathbf{inr}$, so $\mathbf{in} = [\mathbf{nil}, \mathbf{cons}]$. Given any two functions $\mathbf{c} : 1 \to C$ and $\mathbf{h} : A \times C \to C$, the catamorphism $\mathbf{f} = [\mathbf{c}, \mathbf{h}] : \mathbf{List}(A) \to C$ is the unique solution of the equation system:

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

where $\mathbf{f} = \mathbf{foldr}(\mathbf{c}, \mathbf{h})$. Having this the initial algebra is presented with functor $\mu(1 + A \times X)$ and morphisms sum $[1 \to \mathbf{List}(A), A \times \mathbf{List}(A) \to \mathbf{List}(A)]$ as catamorphism. Using this encdoding the base library of List will have following form:

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

```
data list: (A: *) → * :=
     (nil: list A)
     (cons: A → list A → list A)
```

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

```
record lists: (A B: *) :=
       (len: list A → integer)
       ((++): list A → list A → list A)
       (map: (A → B) → (list A → list B))
       (filter: (A → bool) → (list A → list A))
```

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

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.

2.4.3 Імпредикативне кодування

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 [?] and [?]. 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
```

2.5 Гомотопічна система

2.5.1 Синтаксис

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

Here := (definition), + (disjoint sum), #empty, #nat, #list are parts of BNF language and |, :, *, ⟨, ⟩, (,), =, \, /, -, →, 0, 1, @, [,], **module**, **import**, **data**, **split**, **where**, **comp**, **fill**, **Glue**, **glue**, **unglue**, **.1**, **.2**, and , are terminals of type checker language. This language includes inductive types, higher inductive types and gluening operations needed for both, the constructive homotopy type theory and univalence. All these concepts as a part of the languages will be described in the upcoming Issues II—V.

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

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

data HTS (A: U)
   = parent (p: Inductive (HTS A))
   | path (a b: A)
   | pathLam (n: nat) (a b: A)
   | pathApp (f: alg) (a b: A)
   | comp (a b: A)
   | fill (a b c: A)
   | glue (a b c: A)
   | glueElem (a b: A)
   | unglueElem (a b: A)
```

2.6 Гомотопічна базова бібліотека системи доведення

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

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

2.6.1 Основа

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

2.6.2 Математика

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

[9] `https://github.com/groupoid/om`
[10] `https://github.com/groupoid/infinity`

Глава 3

ОСНОВА

3.1 Інтерналізація теорії типів

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

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

3.1.0.1 Теорія типів

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

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

The most interesting are Id types. Id types were added in [1]1984

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

: Interpretations correspond to mathematical theories Таблица 3.1

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

while original MLTT was introduced in [2]1972. Predicative Universe Hierarchy was added in [3]1975. While original MLTT contains Id types that preserve uniquness of identity proofs (UIP) or eta-rule of Id type, HoTT refutes UIP (eta rule desn't hold) and introduces univalent heterogeneous Path equality ([4]∞-Groupoid interpretation). Path types are essential to prove computation and uniquness rules for all types (needed for building signature and terms), so we will be able to prove all the MLTT rules as a whole.

3.1.0.2 Інтерпретації

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

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

[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.

Logical or Type-theoretical interpretation

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

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

Categorical or Topos-theoretical interpretation

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

Homotopical interpretation

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

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

Set-theoretical interpretation

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

3.1.1 Типи Π, Σ, Path

3.1.1.1 Π-тип

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

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

Теоретико-типова інтерпретація

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

Визначення 1. (Π-Formation).

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

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

Визначення 2. (Π-Introduction).

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

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

Визначення 3. (Π-Elimination).

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

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

Теорема 1. (Π-Computation).

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

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

Теорема 2. (Π-Uniqueness).

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

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

Категоріальна інтерпретація

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

Визначення 4. (Dependent Product). The dependent product along morphism $g : B \to A$ in category C is the right adjoint $\Pi_g : C_{/B} \to C_{/A}$ of the base change functor.

Визначення 5. (Space of Sections). Let $\mathbf{H}$ be a $(\infty, 1)$-topos, and let $E \to B : \mathbf{H}_{/B}$ a bundle in $\mathbf{H}$, object in the slice topos. Then the space of sections $\Gamma_\Sigma(E)$ of this bundle is the Dependent Product:

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

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

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

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

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

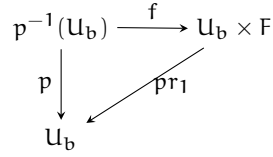
Визначення 6. (Section). A section of morphism $f : A \to B$ in some category is the morphism $g : B \to A$ such that $f \circ g : B \xrightarrow{g} A \xrightarrow{f} B$ equals the identity morphism on B.

Гомотопічна інтерпретація

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

Визначення 7. (Fiber). The fiber of the map $p : E \to B$ in a point $y : B$ is all points $x : E$ such that $p(x) = y$.

Визначення 8. (Fiber Bundle). The fiber bundle $F \to E \xrightarrow{p} B$ on a total space E with fiber layer F and base B is a structure (F, E, p, B) where $p : E \to B$ is a surjective map with following property: for any point $y : B$ exists a neighborhood U_b for which a homeomorphism $f : p^{-1}(U_b) \to U_b \times F$ making the following diagram commute.



Визначення 9. (Cartesian Product of Family over B). Is a set F of sections of the bundle with elimination map $app : F \times B \to E$ such that

$$F \times B \xrightarrow{app} E \xrightarrow{pr_1} B \tag{3.1}$$

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

Визначення 10. (Trivial Fiber Bundle). When total space E is cartesian product $\Sigma(B, F)$ and $p = pr_1$ then such bundle is called trivial $(F, \Sigma(B, F), pr_1, B)$.

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

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

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

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

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

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

3.1.1.2 Σ-тип

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

3.1.1.3 Теоретико-типов інтерпретація

Визначення 11. (Σ-Formation).

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

Визначення 12. (Σ-Introduction).

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

Визначення 13. (Σ-Elimination).

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

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

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

Теорема 8. (Σ-Computation).

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

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

Теорема 9. (Σ-Uniqueness).

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

3.1.1.4 Категоріальна інтерпретація

Визначення 14. (Dependent Sum). The dependent sum along the morphism $f : A \to B$ in category C is the left adjoint $\Sigma_f : C_{/A} \to C_{/B}$ of the base change functor.

3.1.1.5 Теоретико-типова інтерпретація

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

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

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

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

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

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

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

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

3.1.1.6 Path-тип

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

[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**

3.1.1.7 Кубічна інтерпретація

Визначення 15. (Path Formation).

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

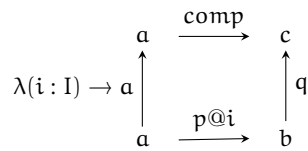
Визначення 16. (Path Reflexivity). Returns an element of reflexivity path space for a given value of the type. The inhabitant of that path space is the lambda on the homotopy interval $[0, 1]$ that returns a constant value a. Written in syntax as |<i>a| which equals to $\lambda\ (i : I) \to a$.

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

Визначення 17. (Path Application). 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
```

Визначення 18. (Path Composition). Composition operation allows to build a new path by given to paths in a connected point.

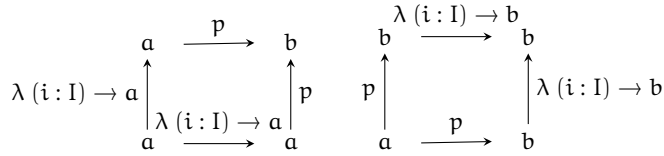



```
composition (A: U) (a b c: A) (p: Path A a b) (q: Path A b c)
          : Path A a c = comp (<i>Path A a (q@i)) p []
```

Теорема 14. (Path Inversion).

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

Визначення 19. (Connections). Connections allows you to build square with given only one element of path: i) $\lambda\ (i, j : I) \to p\ @\ min(i, j)$; ii) $\lambda\ (i, j : I) \to p\ @\ max(i, j)$.




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

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

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

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

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

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

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

3.1.1.8 Теоретико-типова інтерпретація

Визначення 20. (Singleton).

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

Теорема 17. (Singleton Instance).

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

Теорема 18. (Singleton Contractability).

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

Теорема 19. (Path Elimination, Diagonal).

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

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

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

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

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

Теорема 22. (Path Computation).

```
trans_comp (A: U) (a: A)
  : Path A a (trans A A (<_> A) a)
  = fill (<i> A) a []
subst_comp (A: U) (P: A -> U) (a: A) (e: P a)
  : Path (P a) e (subst A P a a (refl A a) e)
  = trans_comp (P a) e
J_comp (A: U) (a: A) (C: (x: A) -> Path A a x -> U) (d: C a (refl A a))
  : Path (C a (refl A a)) d (J A a C d a (refl A a))
  = subst_comp (singl A a) T (eta A a) d where T (z: singl A a)
  : U = C a (z.1) (z.2)
```

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

3.1.1.9 Групоїдна інтерпретація

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

3.1.2 Всесвіти

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

Визначення 21. (Terms). Point in initial object of language AST inductive definition is called a term. If type theory or language is defined as an inductive type (AST) then the term is defined as its instance.

Визначення 22. (Sorts). N-indexed set of universes $\mathcal{U}_{n \in \mathbb{N}}$. 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.

Визначення 23. (Axioms). The inclusion rules $\mathcal{U}_i : \mathcal{U}_j, i, j \in \mathbb{N}$, that define which universe is element of another given universe. You may attach any rules that joins i, j in some way. Axioms with sorts define universe hierarchy.

Визначення 24. (Rules). The set of landings $\mathcal{U}_i \to \mathcal{U}_j : \mathcal{U}_{\lambda(i,j),i,j\in\mathbf{N}}$, where $\lambda : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$. These rules define term dependence or how we land (in which universe) formation rules in definitions.

Визначення 25. (Predicative hierarchy). If λ in Rules is an uncurried function $\mathbf{max} : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ then such universe hierarchy is called predicative.

Визначення 26. (Impredicative hierarchy). If λ in Rules is a second projection of a tuple $\mathbf{snd} : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ then such universe hierarchy is called impredicative.

Визначення 27. (Definitional Equality). For any $\mathcal{U}_i, i \in \mathbf{N}$ there is defined an equality between its members and between its instances. For all x,y ∈ A, there is defined a x=y. Definitional equality compares normalized term instances.

Визначення 28. (SAR). The universum space is configured with a triple of: i) sorts, a set of universes $\mathcal{U}_{n\in\mathbf{N}}$ indexed over set N; ii) axioms, a set of inclusions $\mathcal{U}_i : \mathcal{U}_j, i, j \in \mathbf{N}$; iii) rules of term dependence universe landing, a set of landings $\mathcal{U}_i \to \mathcal{U}_j : \mathcal{U}_{\lambda(i,j),i,j\in\mathbf{N}}$, where λ could be function $\mathbf{max}$ (predicative) or $\mathbf{snd}$ (impredicative).

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

Приклад 2. ($\mathbf{PTS}^\infty$). SAR = $\{\mathcal{U}_{i\in\mathbf{N}}, \mathcal{U}_i : \mathcal{U}_{j;i<j;i,j\in\mathbf{N}}, \mathcal{U}_i \to \mathcal{U}_j : \mathcal{U}_{\lambda(i,j);i,j\in\mathbf{N}}\}$. Where $\mathcal{U}_i$ is a universe of i-level or i-category in categorical interpretation. The working prototype of $\mathbf{PTS}^\infty$ is given in Addendum I: Pure Type System for Erlang[10].

3.1.3 Контексти

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

Визначення 29. (Empty Context).

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

Визначення 30. (Context Comprehension).

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

[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

Визначення 31. (Context Derivability).

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

3.1.4 Інтерналізація

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

Визначення 32. (MLTT). The MLTT as a Type is defined by taking all rules for Π, Σ and Path types into one Σ telescope or context.

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

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

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

Перевірка в кубічній теорії

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

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

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

3.2 Індуктивні типи

Empty

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

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

Unit

```
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
```

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 bool }
boolEq: b2 = lambda bool (bool -> bool) negation
boolRec (C: U) (f t: C): bool -> C = split { false -> f ; true -> t }
boolInd (C: bool -> U) (f: A false) (t: A true): (n:bool) -> A n
      = split { false -> f ; true -> t }
```

3.2.1 Maybe

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

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

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

3.2.2 Either

either is a representation for sum types or disjunction.

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

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

Tuple

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

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

3.2.3 Nat

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

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

3.2.4 List

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

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

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

3.2.5 Stream

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

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

3.2.6 Fin

fin is the inductive defintion of set with finite elements.

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

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

3.2.7 Vector

vector is the inductive defintion of limited length list.

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

seq — abstract compositional sequences.

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

3.2.8 Моделі та кодування

3.3 Гомотопічна теорія типів

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

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

In the same way as inductive types extends MLTT for inductive programming, the higher inductive types (HIT) extend homotopy type theory for geometry programming. You can directly encode CW-complexes by using HIT. The definition of HIT syntax will be given in the next Issue IV: Higher Inductive Types.

3.3.1 Гомотопії

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

Визначення 33. (Interval). Compact interval.

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

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

[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

Визначення 34. (Interval Split). The convertion function from I to a type of comparison is a direct eliminator of interval. The interval is also known as one of primitive higher inductive types which will be given in the next Issue IV: Higher Inductive Types.

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

Визначення 35. (Homotopy). The homotopy between two function $f, g : X \to Y$ is a continuous map of cylinder $H : X \times I \to Y$ such that

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

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

3.3.2 Групоїдна інтерпретація

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

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)
```

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

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

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

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

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

3.3.3 Функціональна екстенсіональність

Визначення 36. (funExt-Formation)

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

Визначення 37. (funExt-Introduction)

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

Визначення 38. (funExt-Elimination)

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

Визначення 39. (funExt-Computation)

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

Визначення 40. (funExt-Uniqueness)

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

3.3.4 Пулбеки

3.3.5 Фібрації

Визначення 41. (Fibration-1) Dependent fiber bundle derived from Path contractability.

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

Визначення 42. (Fibration-2). Dependent fiber bundle derived from surjective function.

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

Визначення 43. (Fibration-3). Non-dependent fiber bundle derived from fiber truncation.

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

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

Визначення 44. (Fibration-4). Non-dependen fiber bundle derived as pullback square.

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

3.3.6 Еквівалентність

Визначення 45. (Equivalence).

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

Визначення 46. (Surjective).

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

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

Визначення 47. (Injective).

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

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

Визначення 48. (Embedding).

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

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

Визначення 49. (Half-adjoint Equivalence).

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

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

3.3.7 Ізоморфізм

Визначення 50. (iso-Formation)

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

Визначення 51. (iso-Introduction)

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

Визначення 52. (iso-Elimination)

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

Визначення 53. (iso-Computation)

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

Визначення 54. (iso-Uniqueness)

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

3.3.8 Унівалентність

Визначення 55. (uni-Formation)

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

Визначення 56. (uni-Introduction)

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

Визначення 57. (uni-Elimination)

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

Визначення 58. (uni-Computation)

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

Визначення 59. (uni-Uniqueness)

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

3.3.9 Простори петель

Визначення 60. (Pointed Space). A pointed type (A, a) is a type $A : U$ together with a point $a : A$, called its basepoint.

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

Визначення 61. (Loop Space).

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

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

Визначення 62. (n-Loop Space).

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

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

3.3.10 Обчислення гомотопічних груп

Визначення 63. (n-th Homotopy Group of m-Sphere).

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

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

Теорема 24. $(\Omega(S^1) = \mathbb{Z})$.

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

loopS1 : U = Path S1 base base

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

decode : (x:S1) -> helix x -> Path S1 base x = split
  base -> loopIt
  loop @ i -> rem @ i where
    p : Path U (Z -> loopS1) (Z -> loopS1)
      = <j> helix (loop1@j) -> Path S1 base (loop1@j)
    rem : PathP p loopIt loopIt
      = corFib1 S1 helix (\(x:S1)->Path S1 base x) base
        loopIt loopIt loop1 (\(n:Z) ->
        comp (<i> Path loopS1 (oneTurn (loopIt n))
             (loopIt (testIsoPath Z Z sucZ predZ
                      sucpredZ predsucZ n @ i)))
             (<i>(lem1It n)@-i) [])

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

3.4 Вищі індуктивні типи

CW-complexes are fundamental objects in homotopy type theory and even included inside cubical type checker in a form of higher

(co)-inductive types (HITs). Just like regular (co)-inductive types could be described as recur- sive terminating (well-founded) or non-terminating trees, higher inductive types could be described as CW-complexes. Defining HIT means to define some CW- complex directly using cubical homogeneous composition structure as an ele- ment of initial algebra inside cubical model.

3.4.1 Інтервал

Визначення 64. (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 $\mathfrak{i}0, \mathfrak{i}1 : \mathrm{I}$ to $x, y : A$ one can obtain identity or equality type from classic type theory.

3.4.2 n-Сфера

Визначення 65. (Shperes and Disks). Here are some example of using dimensions to construct spherical shapes.

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


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

3.4.3 Суспензія

Визначення 66. (Suspension).

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

3.4.4 Транкейшин

Визначення 67. (Suspension).

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

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

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

3.4.5 Факторизація

Визначення 68. (Quotient).

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

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

3.4.6 Пушаут

Визначення 69. (Pushout). One of the notable examples is pushout as it's used to define the cell attachment formally, as others cofibrant objects.

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

3.4.7 CW-комплекси

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

Визначення 70. (Attachment). Attaching n-cell to a space X along a map $f : S^{n-1} \to X$ means taking a pushout figure.

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

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

Визначення 71. (CW-Complex). Inductively. The only CW-complex of dimention -1 is $\varnothing$. A CW-complex of dimension $\leqslant n$ on X is a space X obtained by attaching a collection of n-cells to a CW-complex of dimension $n-1$.

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

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

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

3.5 Модальності

3.5.1 Процеси

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

Визначення 72. (Storage). The secure storage based on verified cryptography. NOTE: For simplicity let it be a compatible list.

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

Визначення 73. (Process). The type formation rule of the process is a Σ telescope that contains: i) protocol type; ii) state type; iii) in-memory current state of process in the form of cartesian product of protocol and state which is called signature of the process; iv) monoidal action on signature; v) persistent storage for process trace.

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

Визначення 74. (Spawn). The sole introduction rule, process constructor is a tuple with filled process type information. Spawn is a modal arrow representing the fact that process instance is created at some scheduler of CPU core.

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

Визначення 75. (Accessors). Process type defines following accessors (projections, this eliminators) to its structure: i) protocol type; ii) state type; iii) signature of the process; iv) current state of the process; v) action projection; vi) trace projection.

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

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

Визначення 76. (Receive). The modal arrow that represents sleep of the process until protocol message arrived.

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

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

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

Визначення 77. (Execute). The Execute function is an eliminator of process stream performing addition of a single entry to the secured storage of the process. Execute is a transactional or synchronized version of asynchronous Send.

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

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

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

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

[14] https://n2o.space/articles/streams.htm
[15] https://n2o.space/articles/bpe.htm
[16] https://mqtt.n2o.space/man/n2o_async.htm

Глава 4

МАТЕМАТИКА

4.1 Теорія множин

4.2 Теорія категорій

Визначення 78. (Category Signature). The signature of category is a $\Sigma_{A:\mathcal{U}} A \to A \to \mathcal{U}$ where $\mathcal{U}$ could be any universe. The pr_1 projection is called Ob and pr_2 projection is called $\mathrm{Hom}(a, b)$, where $a, b : \mathrm{Ob}$.

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

Precategory C defined as set of $\mathrm{Hom}_C(a, b)$ where $a, b : \mathrm{Ob}_C$ are objects defined by its id arrows $\mathrm{Hom}_C(x, x)$. Properfies of left and right units included with composition c and its associativity.

Визначення 79. (Precategory). More formal, precategory C consists of the following. (i) A type Ob_C, whose elements are called objects; (ii) for each $a, b : \mathrm{Ob}_C$, a set $\mathrm{Hom}_C(a, b)$, whose elements are called arrows or morphisms. (iii) For each $a : \mathrm{Ob}_C$, a morphism $1_a : \mathrm{Hom}_C(a, a)$, called the identity morphism. (iv) For each $a, b, c : \mathrm{Ob}_C$, a function $\mathrm{Hom}_C(b, c) \to \mathrm{Hom}_C(a, b) \to \mathrm{Hom}_C(a, c)$ called composition, and denoted $g \circ f$. (v) For each $a, b : \mathrm{Ob}_C$ and $f : \mathrm{Hom}_C(a, b)$, $f = 1_b \circ f$ and $f = f \circ 1_a$. (vi) For each $a, b, c, d : A$ and $f : \mathrm{Hom}_C(a, b)$, $g : \mathrm{Hom}_C(b, c)$, $h : \mathrm{Hom}_C(c, d)$, $h \circ (g \circ f) = (h \circ g) \circ f$.

Визначення 80. (Small Category). If for all $a, b : \mathrm{Ob}$ the $\mathrm{Hom}_C(a, b)$ forms a Set, then such category is called small category.

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

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

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

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

4.2.1 (Ко)термінал

Визначення 81. (Initial Object). Is such object Ob_C, that $\Pi_{x,y:\mathrm{Ob}_C} \mathrm{isContr}(\mathrm{Hom}_C(x,y))$.

Визначення 82. (Terminal Object). Is such object Ob_C, that $\Pi_{x,y:\mathrm{Ob}_C} \mathrm{isContr}(\mathrm{Hom}_C(y,x))$.

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

4.2.2 Функтор

Визначення 83. (Category Functor). Let A and B be precategories. A functor $F : A \to B$ consists of: (i) A function $F_{\mathrm{Ob}} : \mathrm{Ob}_h A \to \mathrm{Ob}_B$; (ii) for each $a, b : \mathrm{Ob}_A$, a function $F_{\mathrm{Hom}} : \mathrm{Hom}_A(a,b) \to \mathrm{Hom}_B(F_{\mathrm{Ob}}(a), F_{\mathrm{Ob}}(b))$; (iii) for each $a : \mathrm{Ob}_A$, $F_{\mathrm{Ob}}(1_a) = 1_{F_{\mathrm{Ob}}}(a)$; (iv) for $a, b, c : \mathrm{Ob}_A$ and $f : \mathrm{Hom}_A(a,b)$ and $g : \mathrm{Hom}_A(b,c)$, $F(g \circ f) = F_{\mathrm{Hom}}(g) \circ F_{\mathrm{Hom}}(f)$.

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

4.2.3 Натуральні перетворення

Визначення 84. (Natural Transformation). For functors $F, G : C \to D$, a nagtural transformation $\gamma : F \to G$ consists of: (i) for each $x : C$, a morphism $\gamma_a : \mathrm{Hom}_D(F(x), G(x))$; (ii) for each $x, y : C$ and $f : \mathrm{Hom}_C(x,y)$, $G(f) \circ \gamma_x = \gamma_y \circ F(g)$.

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

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

4.2.4 Розширення Кана

Визначення 85. (Kan Extension).

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

4.2.5 Category Isomorphism

Визначення 86. (Category Isomorphism). A morphism $f : \mathrm{Hom}_A(a, b)$ is an iso if there is a morphism $g : \mathrm{Hom}_A(b, a)$ such that $1_a =_\eta g \circ f$ and $f \circ g =_\epsilon 1_b = g$. "f is iso"is a mere proposition. If A is a precategory and $a, b : A$, then $a = b \to \mathrm{iso}_A(a, b)$ (idtoiso).

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

4.2.6 Резк-поповнення

Визначення 87. (Category). A category is a precategory such that for all $a : \mathrm{Ob}_C$, the $\Pi_{A:\mathrm{Ob}_C} \mathrm{isContr} \Sigma_{B:\mathrm{Ob}_C} \mathrm{iso}_C(A, B)$.

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

4.2.7 Конструкції

4.2.7.1 (Ко)продукти категорій

Визначення 88. (Category Product).

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

4.2.7.2 Обернена категорія

Визначення 89. (Opposite Category). The opposite category to category $\mathbf{C}$ is a category $\mathbf{C}^{\mathbf{op}}$ with same structure, except all arrows are inverted.

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

4.2.7.3 (Ко)слайс категорія

Визначення 90. (Slice Category).

Визначення 91. (Coslice Category).

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

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

4.2.7.4 Універсальна властивість

Визначення 92. (Universal Mapping Property).

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

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

4.2.7.5 Одинична категорія

Визначення 93. (Unit Category). In unit category both $\mathbf{Ob} = \top$ and $\mathbf{Hom} = \top$.

```
unitCat: precategory
```

4.2.8 Приклади

4.2.8.1 Категорія множин

Визначення 94. (Category of Sets).

```
Set: precategory = ((Ob,Hom),id,c,HomSet,L,R,Q) where
  Ob: U = SET
  Hom (A B: Ob): U = A.1 -> B.1
  id (A: Ob): Hom A A = idfun A.1
  c (A B C: Ob) (f: Hom A B) (g: Hom B C): Hom A C
```

```
  = o A.1 B.1 C.1 g f
HomSet (A B: Ob): isSet (Hom A B) = setFun A.1 B.1 B.2
L (A B: Ob) (f: Hom A B): Path (Hom A B) (c A A B (id A) f) f
 = refl (Hom A B) f
R (A B: Ob) (f: Hom A B): Path (Hom A B) (c A B B f (id B)) f
 = refl (Hom A B) f
Q (A B C D: Ob) (f: Hom A B) (g: Hom B C) (h: Hom C D)
 : Path (Hom A D) (c A C D (c A B C f g) h) (c A B D f (c B C D g h))
 = refl (Hom A D) (c A B D f (c B C D g h))
```

4.2.8.2 Категорія функцій

Визначення 95. (Category of Functions over Sets).

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

4.2.8.3 Категорія категорій

Визначення 96. (Category of Categories).

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

4.2.8.4 Категорія функторів

Визначення 97. (Category of Functors).

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

```
L (A B: Ob) (f: Hom A B): Path (Hom A B) (c A A B (id A) f) f = undefined
R (A B: Ob) (f: Hom A B): Path (Hom A B) (c A B B f (id B)) f = undefined
Q (A B C D: Ob) (f: Hom A B) (g: Hom B C) (h: Hom C D)
 : Path (Hom A D) (c A C D (c A B C f g) h)
                  (c A B D f (c B C D g h)) = undefined
```

4.2.9 k-морфізми

Визначення 98. (k-Morphism). The k-morphism is defined as morphism between $(k-1)$-morphism. The base of induction, the 0-morphism is defined as object of 1-category, which is precategory.

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

І так далі.

4.2.10 2-категорія

Визначення 99. (2-Category).

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

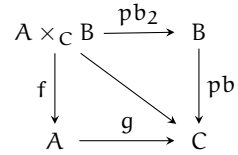
4.2.11 Група Гротендіка

4.3 Теорія топосів

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

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

Визначення 100. (Categorical Pullback). The pullback of the cospan $A \xrightarrow{f} C \xleftarrow{g} B$ is a object $A \times_C B$ with morphisms $pb_1 : \times_C \to A$, $pb_2 : \times_C \to B$, such that diagram commutes:



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

```
homTo  (C: precategory) (X: carrier C): U
  = (Y: carrier C) * hom C Y X
cospan (C: precategory): U
  = (X: carrier C) * (_: homTo C X) * homTo C X
cospanCone (C: precategory) (D: cospan C): U
  = (W: carrier C) * hasCospanCone C D W
cospanConeHom (C: precategory) (D: cospan C)
    (E1 E2: cospanCone C D) : U
  = (h: hom C E1.1 E2.1) * isCospanConeHom C D E1 E2 h
isPullback (C: precategory) (D: cospan C) (E: cospanCone C D) : U
  = (h: cospanCone C D) -> isContr (cospanConeHom C D h E)
hasPullback (C: precategory) (D: cospan C) : U
  = (E: cospanCone C D) * isPullback C D E
```

Визначення 101. (Category Functor). Let A and B be precategories. A functor $F : A \to B$ consists of: (i) A function $F_{Ob} : Ob_h A \to Ob_B$; (ii) for each $a, b : Ob_A$, a function $F_{Hom} : Hom_A(a, b) \to Hom_B(F_{Ob}(a), F_{Ob}(b))$; (iii) for each $a : Ob_A$, $F_{Ob}(1_a) = 1_{F_{Ob}}(a)$; (iv) for $a, b, c : Ob_A$ and $f : Hom_A(a, b)$ and $g : Hom_A(b, c)$, $F(g \circ f) = F_{Hom}(g) \circ F_{Hom}(f)$.

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

Визначення 102. (Terminal Object). Is such object Ob_C, that

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

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

Теорія множин

Here is given the ∞-groupoid model of sets.

Визначення 103. (Mere proposition, PROP). A type P is a mere proposition if for all $x, y : P$ we have $x = y$:

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

Визначення 104. (0-type). A type A is a 0-type is for all $x, y : A$ and $p, q : x =_A y$ we have $p = q$.

Визначення 105. (1-type). A type A is a 1-type if for all $x, y : A$ and $p, q : x =_A y$ and $r, s : p =_{=_A} q$, we have $r = s$.

Визначення 106. (A set of elements, SET). A type A is a SET if for all $x, y : A$ and $p, q : x = y$, we have $p = q$:

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

Визначення 107. `data N = Z | S (n: N)`

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

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

Визначення 108. (Π-Contractability). If fiber is set thene path space between any sections is contractible.

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

Визначення 109. (Σ-Contractability). If fiber is set then Σ is set.

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

Визначення 110. (Unit type, $\mathbf{1}$). The unit $\mathbf{1}$ is a type with one element.

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

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

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

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

4.3.1 Топологічна структура

Визначення 111. (Topology). The topological structure on $\mathcal{A}$ (or topology) is a subset $S \in \mathcal{A}$ with following properties: i) any finite union of subsets of S is belong to S; ii) any finite intersection of subsets of S is belong to S. Subets of S are called open sets of family S.

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

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

4.3.2 Топос Гротендіка

Grothendieck Topology is a calculus of coverings which generalizes the algebra of open covers of a topological space, and can exist

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

on much more general categories. There are three variants of Grothendieck topology definition: i) sieves; ii) coverage; iii) covering families. A category have one of these three is called a Grothendieck site.

Examples: Zariski, flat, étale, Nisnevich topologies.

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

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

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

Визначення 112. (Sieves). Sieves are a family of subfunctors

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

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

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

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

Визначення 113. (Coverage). A coverage is a function assigning to each Ob_C the family of morphisms $\{f_i : U_i \to U\}_{i \in I}$ called covering families, such that for any $g : V \to U$ exist a covering family $\{h : V_j \to V\}_{j \in J}$ such that each composite $h_j \circ g$ factors some f_i:

$$\begin{array}{ccc} V_j & \xrightarrow{k} & U_i \\ \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle f_i} \\ V & \xrightarrow{g} & U \end{array}$$

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

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

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

Визначення 114. (Grothendieck Topology). Suppose category $\mathbf{C}$ has all pullbacks. Since $\mathbf{C}$ is small, a pretopology on $\mathbf{C}$ consists of families of sets of morphisms

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

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

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

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

Визначення 115. (Site). Site is a category having either a coverage, grothendieck topology, or sieves.

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

Визначення 116. (Presheaf). Presheaf of a category $\mathbf{C}$ is a functor from opposite category to category of sets: $\mathbf{C}^{\mathbf{op}} \to \mathbf{Set}$.

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

Визначення 117. (Presheaf Category, PSh). Presheaf category PSh for a site $\mathbf{C}$ is category were objects are presheaves and morphisms are natural transformations of presheaf functors.

Визначення 118. (Sheaf). Sheaf is a presheaf on a site. In other words a presheaf $F : \mathbf{C}^{\mathbf{op}} \to \mathbf{Set}$ such that the cannonical map of inverse limit

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

is an isomorphism for each covering sieve $R \subset \mathbf{Hom}_{\mathbf{C}}(\_, U)$. Equivalently, all induced functions

$$\mathbf{Hom}_{\mathbf{C}}(\mathbf{Hom}_{\mathbf{C}}(\_, U), F) \to \mathbf{Hom}_{\mathbf{C}}(R, F)$$

should be bejections.

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

Визначення 119. (Sheaf Category, Sh). Sheaf category Sh is a category where objects are sheaves and morphisms are natural transformation of sheves. Sheaf category is a full subcategory of category of presheaves PSh.

Визначення 120. (Grothendieck Topos). Topos is the category of sheaves $\mathrm{Sh}(\mathbf{C}, \mathbf{J})$ on a site $\mathbf{C}$ with topology $\mathbf{J}$.

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

Визначення 121. (Geometric Morphism). Suppose that $\mathbf{C}$ and $\mathbf{D}$ are Grothendieck sites. A geometric morphism

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

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

Визначення 122. (Cohesive Topos). A topos E is a cohesive topos over a base topos S, if there is a geometric morphism $(p^*, p_*) : E \to S$, such that: i) exists adjunction $p^! \vdash p_*$ and $p^! \dashv p_*$; ii) p^* and $p^!$ are full faithful; iii) $p_!$ preserves finite products.

This quadruple defines adjoint triple:

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

4.3.3 Елементарний топос

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

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

Визначення 123. (Monomorphism). An morphism $f : Y \to Z$ is a monic or mono if for any object X and every pair of parralel morphisms $g_1, g_2 : X \to Y$ the

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

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

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

Визначення 124. (Subobject Classifier [?]). In category C with finite limits, a subobject classifier is a monomorphism $\mathrm{true} : 1 \to \Omega$ out of terminal object 1, such that for any mono $U \to X$ there is a unique morphism $\chi_U : X \to \Omega$ and pullback diagram:

$$\begin{array}{ccc} U & \xrightarrow{k} & 1 \\ \downarrow & & \downarrow{\scriptstyle true} \\ X\Omega & \xrightarrow{\chi u} & \Omega \end{array}$$

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

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

Визначення 125. (Cartesian Closed Categories). The category C is called cartesian closed if exists all: i) terminals; ii) products; iii) exponentials. Note that this definition lacks beta and eta rules which could be found in embedding **MLTT**.

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

Теорема 28. (Category of Sets is cartesian closed). As you can see from exp and pro we internalize Π and Σ types as SET instances,

the $\mathbf{isSet}$ predicates are provided with contractability. Exitense of terminals is proved by $\mathbf{propPi}$. The same technique you can find in $\mathbf{MLTT}$ embedding.

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

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

Визначення 126. (Elementary Topos). Topos is a precategory which is cartesian closed and has subobject classifier.

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

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

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

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

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

Conclusion

We gave here constructive definition of topology as finite unions and intersections of open subsets. Then make this definition categorically compatible by introducing Grothendieck topology in

three different forms: sieves, coverage, and covering families. Then we defined an elementary topos and introduce category of sets, and proved that Set is cartesian closed, has object classifier and thus a topos.

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

4.4 Алгебраїчна топологія

4.4.1 Теорія груп

4.4.2 Простори

4.4.2.1 Сімпліціальні

4.4.2.2 CW-комплекси

4.4.3 Теорія (Ко)Гомотопій

4.4.4 Теорія (Ко)Гомологій

4.5 Диференціальна геометрія

4.5.1 V-многовиди

4.5.2 G-структури

4.5.3 H-простори

Бібліографія

[1] S.MacLane Categories for the Working Mathematician 1972

[2] W.Lawvere Conceptual Mathematics 1997

[3] P.Curien Category theory: a programming language-oriented introduction 2008

[4] P.Martin-Löf Intuitionistic Type Theory 1984

[5] T.Coquand The Calculus of Constructions. 1988

[6] E.Meijer Henk: a typed intermediate language 1997

[7] H.Barendregt Lambda Calculus With Types 2010

[8] F.Pfenning Inductively defined types in the Calculus of Constructions 1989

[9] P.Wadler Recursive types for free 1990

[10] N.Gambino Wellfounded Trees and Dependent Polynomial Functors 1995

[11] P.Dybjer Inductive Famalies 1997

[12] B.Jacobs (Co)Algebras) and (Co)Induction 1997

[13] V.Vene Categorical programming with (co)inductive types 2000

[14] H.Geuvers Dependent (Co)Inductive Types are Fibrational Dialgebras 2015

[15] T.Streicher A groupoid model refutes uniqueness of identity proofs 1994

[16] T.Streicher The Groupoid Interpretation of Type Theory 1996

[17] B.Jacobs Categorical Logic and Type Theory 1999

[18] S.Awodey Homotopy Type Theory and Univalent Foundations 2013

[19] S.Huber A Cubical Type Theory 2015

[20] A.Joyal What is an elementary higher topos 2014

[21] A.Mortberg Cubical Type Theory: a constructive univalence axiom 2017