Дисертація

Система верифікації програмного забезпечення

Рукопис на здобуття ступеня доктора філософії

Максим Сохацький, Павло Маслянко

Зміст

Глава 1

Вступ

Присвячується Маші та Міші

У цій главі дамо тему, предмет та мету роботи, проведемо огляд існуючих рішень у цій області, та дамо опис структури цієї роботи.

1.1 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.

1.2 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 Аеро-

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

космічного 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.

1.3 Формальна специфікація

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

Існують два фундаментальні підходи до формальних специфікацій: 1) Аглебраїчний підхід, де система описується в термінах операцій, та відношень між ними, 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.

Найбільш стандартизована та прийнята в обсласті формальної верифікації — це нотація Z[3] (Spivey, 1992), приклад модельноорієнтовоної 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.

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

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

[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

(Hoare, 1985), CCS[6] (Milner, 1971), Actor Model, Reactive Streams, etc.

1.4 Формальні методи верифікації

1.4.1 Системи верифікації

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

1.4.2 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.

1.4.3 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).

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

1.5 Історія систем доведення теорем

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).

1.6 Обмеження

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 про-

грама завершиться, потрібно звужувати клас програм, якщо формальні специфікації містять такі властивості.

1.7 Об'єкт 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.

1.8 Мотивація

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.

1.9 Предмет дослідження

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.

1.10 Завдання дослідження

Побудова концептуальної моделі системи доведення теорем та імплементація мови програмування, яка релізує логічну семантику. Основні моделі тут формалізуються в теорії типів.

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 верицікації.

1.11 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 $[0, 1]$; 5) уніфікована базова бібліотека.

1.12 Метематичне забезпечення

1.12.1 Інтуіціоністична теорія типів Мартіна-Льофа

Пер Мартін-Льоф в 1972 році запропонував Π, Σ та Id у якості основних фундаментальних типів. З тих пір усі сучасні системи типів для пруверів побувані наслідуючи цю модель (MLTT, теорія типів Мартіна-Льофа). Було показано, що мова такої системи типів є внутрішнією мовою локальних декартово-замкнених категорій. Π та Σ кодують безпосередньо логічні кванторти $\forall$ та $\exists$, а тип $\to$ є частковим випадком Π-типу, коли вираз B не залежить від x.

Аксіоми:

$$\frac{\Gamma\, x : A \vdash B : \mathrm{Type} \quad \Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash \Pi(x : A) \to B(x) : \mathrm{Type}} \tag{Π}$$

$$\frac{\Gamma\, x : A \vdash B : \mathrm{Type} \quad \Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash \Sigma(x : A) \times B(x) : \mathrm{Type}} \tag{Σ}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : A \quad \Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash \mathrm{Id}_A(a, b)} \tag{Id}$$

Теореми:

рефлексивність	:	$\mathrm{Id}_A(a, a)$
підстановка	:	$\mathrm{Id}_A(a, a') \to B(x = a) \to B(x = a')$
симетричність	:	$\mathrm{Id}_A(a, b) \to \mathrm{Id}_A(b, a)$
транзитивність	:	$\mathrm{Id}_A(a, b) \to \mathrm{Id}_A(b, c) \to \mathrm{Id}_A(a, c)$
конгруентність	:	$(f : A \to B) \to \mathrm{Id}_A(x, x') \to \mathrm{Id}_B(f(x), f(x'))$

1.12.2 Теорія категорій

Теорія категорій широко застосовується як інструмент для математиків у тому числі і при аналізі програмного забезпечення. Теорію категорій можна вважати абстрактною алгеброю функцій. Дамо конструктивне визначення категорії. Категорії (програми) визначаються переліком своїх об'єктів (типів) та своїх морфізмів (функцій), а також бінарною операцією композиції, що задовольняє закону асоціативності, та з тотожнім морфізмом (тотжньою функцією — одиницею) який існує для кожного об'єкту (типу) категорії. Аксіоми формації об'єктів не приводяться та автопостулуються в нижніх аксіомах. Поки що тут буде визначатися тільки композиція морфізмів. Об'єкти A та B морфізма $f : A \to B$ називаються домен та кодомен відповідно. Композиція є фундаментальною властивістю морфізмів.

Інтро аксіоми – асоціативність композиції та права і ліва композиції одиниці показують, що категорії є типізованими моноїдами, що складаються з морфізмів та операції композиції. Є різні мови, у тому числі і графічні, представлення категорної семантики, однак у цій роботі ми будемо використовувати теоретико-логічні формулювання.

Аксіоми:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash g \circ f : A \to C}$$

$$\frac{}{\Gamma \vdash \mathrm{id}_A : A \to A}$$

$$\frac{\Gamma \vdash f : B \to A \qquad \Gamma \vdash g : C \to B \qquad \Gamma \vdash h : D \to C}{\Gamma \vdash (f \circ g) \circ h = f \circ (g \circ h) : D \to A}$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \circ \mathrm{id}_A = f : A \to B}$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash \mathrm{id}_B \circ f = f : A \to B}$$

Алгебраїчні типи даних

Після операції композиції, як способу конструювання нових об'єктів за допомогою морфізмів далі йде операція конструювання добутка двох об'єктів певної категорії, разом з добутком морфізмів зі спільним доменом, необхідних для визначення декартового добутка $A \times B$.

Це є внутрішня мова декартової категорії, у якій для будь яких двох доменів існує їх декартова сума (кодобутку) та декартовий добуток (косума, кортеж), за допомогою яких конструюються суми-протоколи та добутки-повідмоення, а також існує $\bot$ тип-термінал, та $\top$ тип-котермінал. Термінальними типами зручно термінувати рекурсивні типи даних, такі як списки. Ми будемо розглядати тільки категорії які маються добутки та суми.

Добуток має природні елімінатори π зі спільним доменом, які є морфізмами-проекціями об'єктів добутка. Сума має оберненені елімінатори σ зі спільним кодоменом. Як видно добуток є дуальний до суми з точністю до направлення стрілок, таким чином елімінатори π та σ є оберненими, тобто $\pi \circ \sigma = \sigma \circ \pi = \mathrm{id}$.

Аксіоми:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash g : A \to C \qquad \Gamma \vdash B \times C}{\Gamma \vdash \langle f, g \rangle : A \to B \times C}$$

$$\frac{}{\Gamma \vdash \top}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash a \mid b : A + B}$$

$$\frac{\Gamma\, x : A \times B}{\Gamma \vdash \pi_1 : A \times B \to A; \Gamma \vdash \pi_2 : A \times B \to B}$$

$$\frac{\Gamma\, x : A + B}{\Gamma \vdash \sigma_1 : A \to A + B; \Gamma \vdash \sigma_2 : B \to A + B}$$

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

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash g : A \to C \qquad \Gamma \vdash B + C}{\Gamma \vdash [f, g] : A \to B + C}$$

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

Теореми:

$$(f \circ g) \circ h = f \circ (g \circ h)$$
$$f \circ \mathrm{id} = f$$
$$\mathrm{id} \circ f = f$$
$$\pi_1 \circ \langle f, g \rangle = f$$
$$\pi_2 \circ \langle f, g \rangle = g$$
$$\langle f \circ \pi_1, f \circ \pi_2 \rangle = f$$
$$\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle$$
$$\langle \pi_1, \pi_2 \rangle = \mathrm{id}$$

1.12.3 Лямбда числення

Будучи внутрішньою мовою декартово-замкненої категорії лямбда числення окрім змінних та констант у вигляді термів пропонує операції абстракції та аплікації, що визначає достатньо лаконічну та потужну структуру обчислень з функціями вищих порядків, та метатипизаціями, такими як System F, яка була запропонована вперше Робіном Мілнером в мові ML, та зараз присутня в більш складних типоих системах, таких як System Fω, та системах Haskell та Scala.

З категоріальної точки зору експоненти $f : A^B$ є аналогами фукціональних просторів $f : B \to A$. Так як ми вже визначили добутки та термінали, то ми можемо визначити і експоненти, опускаючи усі категоріальні подробиці ми визначимо конструювання функції (операція абстракції), яка параметризується змінною x у середовищі Γ; та її елімінатора – операції аплікації функції до аргументу. Так визначаєьтся декартово-замкнена категорія. Визначається також рекурсивний механізм виклику функції з довільною кількістю аргументів.

Аксіоми:

$$\frac{\Gamma\, x : A \vdash M : B}{\Gamma \vdash \lambda\, x\, .\, M : A \to B}$$

$$\frac{\Gamma\, f : A \to B \qquad \Gamma\, a : A}{\Gamma \vdash \mathbf{apply}\ f\ a\ : (A \to B) \times A \to B}$$

$$\frac{\Gamma \vdash f : A \times B \to C}{\Gamma \vdash \mathbf{curry}\ f : A \to (B \to C)}$$

Теореми:

$$\begin{gathered}
\mathbf{apply} \circ \langle (\mathbf{curry}\ f) \circ \pi_1, \pi_2 \rangle = f \\
\mathbf{curry}\ \mathbf{apply} \circ \langle g \circ \pi_1, \pi_2 \rangle) = g \\
\mathbf{apply} \circ \langle \mathbf{curry}\ f, g \rangle = f \circ \langle \mathbf{id}, g \rangle \\
(\mathbf{curry}\ f) \circ g = \mathbf{curry}\ (f \circ \langle g \circ \pi_1, \pi_2 \rangle) \\
\mathbf{curry}\ \mathbf{apply} = \mathbf{id}
\end{gathered}$$

Об'єкти : $\top \mid \bot \mid \to \mid \times \mid +$
Морфізми : $\mathbf{id} \mid f \circ g \mid [f, g] \mid \langle f, g \rangle \mid \mathbf{apply} \mid \lambda \mid \mathbf{curry}$

1.12.4 Індуктивні типи

Системи з залежними типами як верифікаційні математичні формальні моделі для доведення корректності. Система Σ та Π типів, як кванторів існування та узагальнення. Системи Mizar, Coq, Agda, Idris, F*, Lean. Ми будемо використовувати cubicaltt, Coq та Lean для доведення MLTT моделей.

Розбудовуючи певний фреймворк чи систему конструктивними методами так чи інакше доведеться зробити певний вибір у мові та способі кодування. Так при розробці теорії абстрактної алгебри в Coq були використані поліморфні індуктивні структури. Однак Agda та Idris використовують для побудови алгебраїчної теорії типи класів, а у Idris взагалі відсутні поліморфірні індуктивні структури та коіндуктивні структури. В Lean теж відсутні коіндуктивні структури проте повністю реалізована теорія HoTT на нерекурсивних поліморфних структурах що обєднує основні чотири класи математичних теорій: логіка, топологія, теорія множин, теорія типів. Як було показано Стефаном Касом, одна з стратегій імплементації типів класів — це використання поліморфних структур.

$$\frac{A : \mathrm{Type} \quad x : A \quad B(x) : \mathrm{Type}}{W(x : A) \to B(x) : \mathrm{Type}} \qquad (W\text{-formation})$$

$$\frac{a : A \quad t : B(a) \to W}{\mathrm{sup}(a, t) : W} \qquad (W\text{-intro})$$

$$\frac{w : W \vdash C(w) : \mathrm{Type} \quad x : A,\ u : B(x) \to W, \quad v : \Pi(y : B(x)) \to C(u(y)) \vdash c(x, u, v) : C(\mathrm{sup}(x, u))}{w : W \vdash w\mathrm{rec}(w, c) : C(w)} \qquad (W\text{-elimination})$$

Числення процесів

Теорія π-числення процесів Роберта Мілнера є основним формалізмом обчислювальної теорії розподілених систем та її імплементації. З часів виникнення CSP числення розробленого Хоаром, Мілнеру вдалося значно розширити та адаптувати теорію до сучасних телекомунікаційних вимог, як наприклад хендовери в мобільних мережах. Основні теорми в моделі π-числення стосуються непротиречивості та неблокованості у синхронному виконанні мобільних процесів. Так як сучасний Web можно розглядати як телекомунікаційну систему, тому у розробці додатків можна покладатися у тому числі і на такі моделі як π-числення. Також ми анонсуємо процес як фундаменльний тип даних, подібний до функції але який здатний тримати певний стан у вигляді типа коротежа та є морфізмом-одиницею типу свого стану.

$$\frac{\Gamma \vdash E, \Sigma, X \qquad \Gamma \vdash \mathbf{action} : \Sigma \times X \to \Sigma \times X}{\Gamma \vdash \mathbf{spawn\ action} : \pi_\Sigma}$$

$$\frac{\Gamma \vdash \mathbf{pid} : \pi_\Sigma \qquad \Gamma \vdash \mathbf{msg} : \Sigma}{\Gamma \vdash \mathbf{join\ msg\ pid} : \Sigma \times \pi_\Sigma \xrightarrow{\bullet} \Sigma;\ \Gamma \vdash \mathbf{send\ msg\ pid} : \Sigma \times \pi_\Sigma \to \Sigma}$$

$$\frac{\Gamma \vdash L : A + B, R : X + Y \qquad \Gamma \vdash M : A \to X, N : B \to Y}{\Gamma \vdash \mathbf{receive}\ L\ M\ N : L \xrightarrow{\bullet} R}$$

Алгебра процесів визначає базові операції мультиплексування двох чи декількох протоколів в рамках одного процесу (добуток), а також паралельного запуску процесів (сума).

$$\begin{aligned} \oplus &\quad : \quad \pi \parallel \pi \\ \otimes &\quad : \quad \pi \mid \pi \end{aligned}$$

1.13 Теоретична частина. Дослідження систем типів

Головна ідея цієї роботи – побудова гнучкої сучасної мови, яка здатна була би обслуговувати академію та виробництво, тобто у якій можно було би створювати моделі, тут же у цій мові їх доводити, та екстрагувати код через оптимізоване ядро у інтерпритатор чи мови які продукують машинний код.

Працюючи над розподіленими системами, системами зберігання даних та системами обробки тензорних масивів, автором було виявлено глибокий зв'язок Pi-числення процесів (Erlang, Ling) і Stream-числення для обробки тензорних масивів (Futhark, Spiral). Тому ми почали займатися дослідженням гнучкої типової системи яка би могла поєднувати різні моделі мов з одним уніфікованим MLTT ядром.

В ході дослідження були виявлені основні типові системи, або мовні рівні, які пропонується сприймати як протоколи, які можуть підключатися додаючи до системи нові мовні рівні.

1.13.1 PTS системи

З тих часів, як Кокуанд відкрив числення конструкцій, та Берендрехт систематизував його варіації, теорія чистих типових систем (Pure Type Systems, PTS) стала вже достатньо розробленою. Також вона відома як теорія з одніє аксіомою — Π-типом MLTT теорії без Σ та Id типа. Ця теорія репрезентую функціональну повноту згідно Герміди-Якобса.

```
name: U = list nat

data O₁ = star (n: nat)
        | var (n: name)
        | app (f a: O₁)
        | lambda (x: name) (d c: O₁)
        | arrow (d c: O₁)
        | pi (x: name) (d c: O₁)
```

Такі системи є простими, проте зрозумілими та досить потужними, аби реалізувати на такій системі системі System F бібліотеку. Нами була вибрана система PTS∞, яка підтримує нескінченну кількість всесвітів, що унеможливлює парадокси Хуркенса-Рассела-Жирара і має два режими: предикативний та імпредикативний. Ця система типів представлена у нашій роботі як мова OM[7], яка виступає ядром системи доведення, верифікації та екстрагування.

[7] http://github.com/groupoid/om

1.13.2 MLTT системи

Для доведення теорем необхідним є Σ-тип, а також рівність на твердженнях. Саме тому теорія типів Мартіна-Льофа з самого початку була облаштова цими конструціями. В MLTT Σ-тип репрезентує контекстуальну повноту згідно Герміди-Якобса. Σ-тип необхідний також для моделювання алгебраїчних структурносіїв.

```
data O₂ = O₁
        | sigma (n: name) (a b: O₂)
        | pair (a b: O₂)
        | fst (p: O₂)
        | snd (p: O₂)

data O₃ = O₂
        | id (a b: O₃)
        | idPair (a b: O₃)
        | idJ (a b c d e: O₃)
```

1.13.3 Індуктивні системи

В процесі глибшого дослідження індукції в MLTT теорії виникла теорія індуктивних типів, поліноміальних функторів у застосуванні до теорії типів (або рекурсивних дерев з визначеною базою рекурсії, W-типи). В мовах програмування вони відомі як data типи. Насправді індуктивні типи можуть бути закодовані в PTS системах за допомогою кодування Черча-Бома-Берардуччі, або в інших системах кодувань (Парігот, Скот, CPS, Ламбек). Рекурсивні дерева без бази рекурсії називаються ко-індуктивними типами, та часто можуть бути представлені як record типи, або рекурсивні record типи.

```
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 O₄ = O₃
        | sum  (n: name) (t: tele O₄) (labels:   list (label O₄))
        | case (n: name) (t: ind)      (branches: list (branch O₄))
        | ctor (n: name)               (args:     list O₄)
```

1.13.4 HTS системи

Багаточисельні фундаменальні дослідження рівності в MLTT теорії привели до тактування — як топологічних просторів. Так в чекарах виник Path-тип, розширений багатовимірний варінт Id-типу з оригінальної теорії Мартіна-Льофа. Це тип моделює відрізок $[0, 1]$ разом з алгеброю де Морана на цьому інтервалі. Ця система типів необхідна для доведення гомотопічних теорем та при роботі з вищими індуктивними типами, за домогою яких кодуються топологічні об'єкти: багатовимірник відрізок, багатовимірна сфера, топологічні операції. Такі системи типів називаються гомотопічними (Homotopy Type System, HTS) та вперше були запропоновані Воєводським.

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

data O₅ = O₄
         | path (a b: O₅)
         | pathLam (n: name) (a b: O₅)
         | pathApp (f: alg) (a b: O₅)
         | comp (a b: O₅)
         | fill (a b c: O₅)
         | glue (a b c: O₅)
         | glueElem (a b: O₅)
         | unglueElem (a b: O₅)
```

1.14 Практична частина. Дослідження операційних середовищ 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.

1.15 Продукти дослідження

1.15.1 Інтерпретатор O на Rust для L4

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.

1.15.2 Коіндукція на Coq з екстрактом в 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.

1.15.3 Екстракція в Erlang та O з OM

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.

Idris та PureScript разом з Erlang

Також був досліджений спосіб естракції Erlang програм з мови програмування Idris, розглянутий протокол передачі термів-теорем в Lean, та екстракція з PureScript в Erlang.

1.16 Структура роботи

Глава 2

Концептуальна модель та структура роботи

2.1 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.

2.2 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.

2.3 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.

2.4 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).

2.4.1 PTS синтаксиси

Мінімальне ядро з однією аксіомою сприймає декілька лямбда ситаксисів. Перший синтаксис сумісний з системою програмування $\mathbf{morte}$[1], та походить від неї. Інший синтаксис сумісний з синтаксисом $\mathbf{cubical}$[2]. Планувалося також підтримати синтаксис $\mathbf{caramel}$[3].

```
data PTS (A: U)
   = star (n: nat)
   | var (n: nat)
   | app (f a: A)
   | lambda (x: nat) (d c: A)
   | arrow (d c: A)
   | pi (x: nat) (d c: A)
```

2.4.2 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)
```

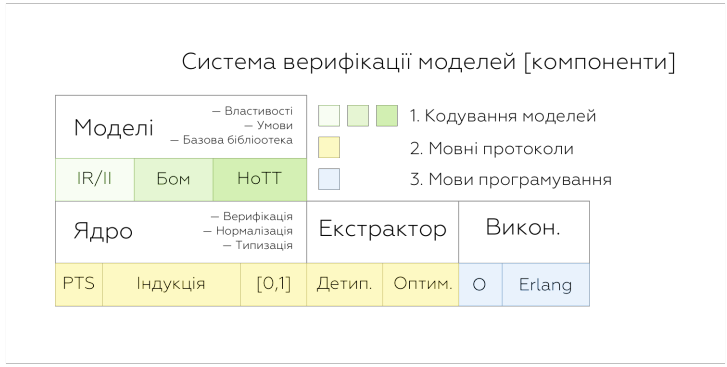
2.4.3 Інші синтаксиси та мови

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

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

[1] http://github.com/Gabriel439/Haskell-Morte-Library
[2] http://github.com/mortberg/cubicaltt
[3] https://github.com/MaiaVictor/caramel

```
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)
```



2.5 Вихідні синтаксиси. Екстракт

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.

2.6 Динаміка. Морфізми

Кожна мова програмування може бути доменом або кодоменом морфізмів в категорії мов програмування. На малюнку зображена композиція морфізмів (верифікації та екстрагування) в maybe-категорії (побудованої maybe функтором) у вигляді BPMN процесу.

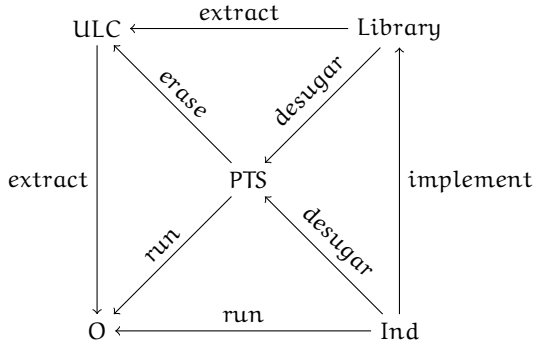
```
process (p: maybe PTS): maybe Erlang
  = extract1 (erase (opt (norm (type p))))
```

Верифікація моделі [процес]

Специфік.	Верифікація		Результ.	Екстракція
Властивості	Нормалізація	=	Конструкція	Детипізація
Гран. Умови	Підст.	Типизація	Помилка	Оптмизікація

```
type (A: maybe PTS): maybe PTS
norm (A: maybe PTS): maybe PTS
erase (A: maybe PTS): maybe ULC
opt (A: maybe PTS): maybe PTS
extract1 (A: maybe ULC): maybe Erlang
extract2 (A: maybe ULC): maybe O
extract3 (A: maybe ULC): maybe C++
extract4 (A: maybe Inductive): maybe Erlang
```

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.



$\mathrm{PTS} : \mathrm{cat}(\mathrm{PTS}, \mathrm{hom})$	—	PTS категорія
$\mathrm{Ind} : \mathrm{cat}(\mathrm{Inductive}, \mathrm{hom})$	—	Індуктивна категорія
$\mathrm{Library} : \mathrm{Inductive}$	—	Базова бібліотека як програма Індиктивної мови
$\mathrm{desugar} : \mathrm{Ind} \to \mathrm{PTS}$	—	Сам прувер є розширенням ядра
$\mathrm{implement} : \mathrm{Ind} \to \mathrm{Library}$	—	Базова бібліотека написана на Індуктивній мові
$\mathrm{lower} : \mathrm{Ind} \to \mathrm{PTS}$	—	Базова бібліотека конвертується в код ядра
$\mathrm{erase} : \mathrm{PTS} \to \mathrm{ULC}$	—	Видаляється інформація про типи, детипізація
$\mathrm{extract} : \mathrm{ULC} \to \mathrm{O}$	—	Запуск на інтерпритаторі

2.7 Операційні семантики

2.8 Мови та Мовні рівні

```
System F-omega
Kind   = * | →    | ···
Con c  =
       | arr( c;  c)  c → c          (Con, Con) Con
       | all  {} (u.c)   (u   ). c (Kind, Con.Con) Con
       | lam  {} (u.c)   (u   ). c (Kind, Con.Con) Con
       | app( c;  c)  c ( c)        (Con, Con) Con
```

2.9 Середовище виконання (ОС)

2.10 Специфікації мов

2.11 Властивості

2.12 Нормалізація та оптимізація

2.13 Екстракція

2.14 Обмеження

2.15 Область застосування

Глава 3

Система з однією аксіомою та її розширення

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 чотири конструкції: формація, інтро, елімінатор та редуктор.

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

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

An intermediate Om language is based on Henk [?] 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.

3.1 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 name
    = list nat

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

3.2 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
```

$$\frac{o : Nat}{Type_o}$$

3.3 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):

```
S    (n : nat) = U n
A₁ (n m : nat) = U n : U m where m > n    - cumulative
R₁ (m n : nat) = U m → U n : U (max m n) - predicative
```

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

$$\frac{i : Nat, j : Nat, i < j}{Type_i : Type_j}$$

$$\frac{i : Nat, j : Nat}{Type_i \to Type_j : Type_{max(i,j)}}$$

3.4 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.

```
A₂ (n : nat) = U n : U (n + 1)   - non-cumulative
R₂ (m n : nat) = U m → U n : U n - impredicative
```

$$\frac{i : Nat}{Type_i : Type_{i+1}} \quad (A_2)$$

$$\frac{i : Nat, \quad j : Nat}{Type_i \to Type_j : Type_j} \quad (R_2)$$

3.5 Система з однією аксіомою

This langauge is called one axiom language (or pure) as eliminator and introduction adjoint functors inferred from type formation rule. The only computation rule of Pi type is called beta-reduction.

```
∀ (x: A) → B x : Type
λ (x: A) → b : B x
f a : B [a/x]
( λ (x: A) → b) a = b[a/x] : B[a/x]
```

$$\frac{x : A \vdash B : Type}{\Pi\,(x : A) \to B : Type} \quad (\Pi\text{-formation})$$

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

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

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

This language could be embedded in itself and used as Logical Framework for the Pi type:

```
record Pi (A: Type) :=
       (intro:  (A → Type) → Type)
       (lambda: (B: A → Type) → pi A B → intro B)
       (app:    (B: A → Type) → intro B → pi A B)
       (applam: (B: A → Type) (f: pi A B) → (a: A) →
                Path (B a) ((app B (lambda B f)) a) (f a))
       (lamapp: (B: A → Type) (p: intro B) →
                Path (intro B) (lambda B (λ (a:A) → app B p a)) p)
```

3.6 Ієрархії

```
dep Arg Out impredicative → Out
dep Arg Out predicative   → max Arg Out

h Arg Out → dep Arg Out om:hierarchy(impredicative)
```

3.7 Перевірка всесвітів

```
star (:*,N) → N
star _      → (:error, "*")
```

3.8 Перевірка Π-типів

```
fun ((:∀,),(I,O)) → true
fun T             → (:error,(:∀,T))
```

3.9 Перевірка змінних

```
var N B        → var N B (proplists:is_defined N B)
var N B true   → true
var N B false  → (:error,("free var",N,proplists:get_keys(B)))
```

3.10 Індекси де Брейна

```
sh (:var,(N,I)),N,P) when I>=P → (:var,(N,I+1))
sh ((:∀,(N,O)),(I,O)),N,P)     → ((:∀,(N,O)),sh I N P,sh O N P+1)
sh ((:λ,(N,O)),(I,O)),N,P)     → ((:λ,(N,O)),sh I N P,sh O N P+1)
sh (Q,(L,R),N,P)               → (Q,sh L N P,sh R N P)
sh (T,N,P)                     → T
```

3.11 Нормалізація

```
norm :none              → :none
norm :any               → :any
norm (:app,(F,A))       → case norm F of
                            ((:λ,(N,O)),(I,O)) → norm (sub O N A)
                            NF → (:app,(NF,norm A)) end
norm (:remote,N)        → cache (norm N [])
norm (:→,       (I,O)) → ((:∀,("_",O)),(norm I,norm O))
norm ((:∀,(N,O)),(I,O)) → ((:∀,(N,O)),  (norm I,norm O))
norm ((:λ,(N,O)),(I,O)) → ((:λ,(N,O)),  (norm I,norm O))
norm T                  → T
```

3.12 Підстановка

```
sub Term Name Value           → sub Term Name Value 0
sub (:→,       (I,O)) N V L → (:→,       sub I N V L,sub O N V L);
sub ((:∀,(N,O)),(I,O)) N V L → ((:∀,(N,O)),sub I N V L,sub O N(sh V N O)L+1)
sub ((:∀,(F,X)),(I,O)) N V L → ((:∀,(F,X)),sub I N V L,sub O N(sh V F O)L)
sub ((:λ,(N,O)),(I,O)) N V L → ((:λ,(N,O)),sub I N V L,sub O N(sh V N O)L+1)
sub ((:λ,(F,X)),(I,O)) N V L → ((:λ,(F,X)),sub I N V L,sub O N(sh V F O)L)
sub (:app,      (F,A)) N V L → (:app,sub F N V L,sub A N V L)
sub (:var,      (N,I)) N V L when I>L → (:var,(N,I-1))
sub (:var,      (N,L)) N V L → V
sub T                  _ _ _ → T.
```

3.13 Рівність за визначенням

```
eq ((:∀,("_",O)), X)    (:→,Y)       → eq X Y
eq (:app,(F1,A1))       (:a,(F2,A2)) → let true = eq F1 F2 in eq A1 A2
eq (:*,N)               (:*,N)       → true
eq (:var,(N,I))         (:var,(N,I)) → true
eq (:remote,N)          (:remote,N)  → true
eq ((:∀,(N1,O)),(I1,O1)) ((:∀,(N2,O)),(I2,O2)) →
   let true = eq I1 I2 in eq O1 (sub (sh O2 N1 0) N2 (:var,(N1,0)) 0)
eq ((:λ,(N1,O)),(I1,O1)) ((:λ,(N2,O)),(I2,O2)) →
   let true = eq I1 I2 in eq O1 (sub (sh O2 N1 0) N2 (:var,(N1,0)) 0)
eq (A,B) → (:error,(:eq,A,B))
```

3.14 Перевірка типів

```
type (:*,N)               _ → (:*,N+1)
type (:v,(N,I))           D → let true = var N D in keyget N D I
type (:#,N)               D → cache type N D
type (:→,(I,O))           D → (:*,h(star(type I D)),star(type O D))
type ((:∀,(N,O)),(I,O))   D → (:*,h(star(type I D)),star(type O [(N,norm I)|D]))
type ((:λ,(N,O)),(I,O))   D → let star (type I D),
                                  NI = norm I in ((:∀,(N,O)),(NI,type(O,[(N,NI)|D])))
type (:a,(F,A))           D → let T = type(F,D),
                                  true = fun T,
                                  ((:∀,(N,O)),(I,O)) = T,
                                  Q = type A D,
                                  true = eq I Q in norm (subst O N A)
```

3.15 Екстракт в платформу Erlang/OTP

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.

Exe Macrosystem

Exe is a general purpose functional language with functors, lambdas on types, recursive algebraic types, higher order functions, corecursion, free monad for effects encoding. It compilers to a small core of dependent type system without recursion called Om. This language intended to be useful enough to encode KVS (database), N2O (web framework) and BPE (processes) applications.

Compiler Passes

The underlying OM typechecker and compiler is a target language for EXE general purpose language.

EXPAND	EXE – Macroexpansion
NORMAL	OM – Term normalization and typechecking
ERASE	OM – Delete information about types
COMPACT	OM – Term Compactification
EXTRACT	OM – Extract Erlang Code

BNF

```
<> ::= #option
[] ::= #list
 I ::= #identifier
 U ::= * < #number >
 O ::= I | ( O ) |
       U | O → O | O O
         | λ ( I : O ) → O
         | ∀ ( I : O ) → O
 L ::= I | L I
 A ::= O | A → A | ( L : O )
 F ::=  | F ( I : O ) | ()
 E ::= O | E data L : A := F
         | E record L : A < extend F > := F
         | E let F in E
         | E case E [ | I O → E ]
         | E receive E [ | I O → E ]
         | E spawn E raise L := E
         | E send E to E
```

3.16 Індуктивні типи

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.

3.17 Поліноміальні функтори

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.

3.18 Кодування List

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 $c : 1 \to C$ and $h : A \times C \to C$, the catamorphism $f = [c, h] : \mathbf{List}(A) \to C$ is the unique solution of the equation system:

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

where $f = \mathbf{foldr}(c, h)$. 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:

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

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

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

```
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))
```

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

3.19 Нормальні форми

List/map

λ (a: *) → λ (b: *) → λ (f: a → b) → λ (xs: ∀ (List: *) → ∀ (Cons: ∀ (head: a) → ∀ (tail: List) → List) → ∀ (Nil: List) → List) → xs (∀ (List: *) → ∀ (Cons: ∀ (head: b) → ∀ (tail: List) → List) → ∀ (Nil: List) → List) (λ (head: a) → λ (tail: ∀ (List: *) → ∀ (Cons: ∀ (head: b) → ∀ (tail: List) → List) → ∀ (Nil: List) → List) → λ (List: *) → λ (Cons: ∀ (head: b) → ∀ (tail: List) → List) → λ (Nil: List) → Cons (f head) (tail List Cons Nil)) (λ (List: *) → λ (Cons: ∀ (head: b) → ∀ (tail: List) → List) → λ (Nil: List) → Nil)

List/filter

(∀ (a: *1) → (∀ (f: (∀ (_:a)→ (∀ (Bool: *1) → (∀ (True: Bool) → (∀ (False: Bool) → Bool))))) → (∀ (xs: (∀ (List: *1) → (∀ (Cons: (∀ (head: a) → (∀ (tail: List) → List))) → (∀ (Nil: List) → List)))) → (∀ (Nil: (∀ (List: *1) → (∀ (Cons: (∀ (head: a) → (∀ (tail: List) → List))) → (∀ (Nil: List) → List)))) → (∀ (List: *1) → (∀ (Cons: (∀ (head: a) → (∀ (tail: List) → List))) → (∀ (Nil: List) → List)))))))

IOI/MkIO

(λ (r: *1) → (λ (s: *1) → (λ (seed: s) → (λ (step: (∀ (_:s)→ (∀ (IOF: *1) → (∀ (PutLine:(∀ (_:(∀ (List: *1) → (∀ (Cons: (∀ (Head: (∀ (Nat: *1) → (∀ (Succ: (∀ (_:Nat)→ Nat)) → (∀ (Zero: Nat) → Nat)))) → (∀ (Tail: List) → List))) → (∀ (Nil: List) → List)))) → (∀ (_:s)→ IOF))) → (∀ (GetLine:(∀ (_:(∀ (_:(∀ (List: *1) → (∀ (Cons: (∀ (Head: (∀ (Nat: *1) → (∀ (Succ: (∀ (_:Nat)→ Nat)) → (∀ (Zero: Nat) → Nat)))) → (∀ (Tail: List) → List))) → (∀ (Nil: List) → List)))) → s)) → IOF)) → (∀ (Pure:(∀ (_:r)→ IOF)) → IOF)))))) → (λ (x: *1) → (λ (k: (∀ (s: *1) → (∀ (_:s)→ (∀ (_:(∀ (_:s)→ (∀ (IOF: *1) → (∀ (PutLine:(∀ (_:(∀ (List: *1) → (∀ (Cons: (∀ (Head: (∀ (Nat: *1) → (∀ (Succ: (∀ (_:Nat)→ Nat)) → (∀ (Zero: Nat) → Nat)))) → (∀ (Tail: List) → List))) → (∀ (Nil: List) → List)))) → (∀ (_:s)→ IOF))) → (∀ (GetLine:(∀ (_:(∀ (_:(∀ (List: *1) → (∀ (Cons: (∀ (Head: (∀ (Nat: *1) → (∀ (Succ: (∀ (_:Nat)→ Nat)) → (∀ (Zero: Nat) → Nat)))) → (∀ (Tail: List) → List))) → (∀ (Nil: List) → List)))) → s)) → IOF)) → (∀ (Pure:(∀ (_:r)→ IOF)) → IOF)))))) → x)))) → (((k s) seed) step)))))))

Глава 4

Базова бібліотека

```
  data Nat: Type :=
       (Zero: Unit → Nat)
       (Succ: Nat → Nat)

  data List (A: Type) : Type :=
       (Nil: Unit → List A)
       (Cons: A → List A → List A)

record list: Type :=
       (len: List A → integer)
       ((++): List A → List A → List A)
       (map: (A,B: Type) (A → B) → (List A → List B))
       (filter: (A → bool) → (List A → List A))

record String: List Nat := List.Nil

  data IO: Type :=
       (getLine: (String → IO) → IO)
       (putLint: String → IO)
       (pure: () → IO)

record IO: Type :=
       (data: String)
       ([>>=]: ...)

record Morte: Type :=
       (recursive: IO.replicateM Nat.Five
                   (IO.[>>=] IO.data Unit IO.getLine IO.putLine))
```

Глава 5

Середовище виконання, Застосування та Висновки

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