

# Дисертація

Система верифікації програмного забезпечення

Рукопис на здобуття ступеня доктора філософії

Максим Сохацький, Павло Маслянко



# Зміст

---

1	Вступ	1
1.1	Системна інженерія та верифікація	1
1.2	Історія систем верифікації	2
1.3	Методи верифікації	3
1.4	Метематичне забезпечення	4
1.4.1	Інтуїціоністична теорія типів Мартіна-Льофа	4
1.4.2	Теорія категорій	5
1.4.3	Алгебраїчні типи даних	6
1.4.4	Лямбда числення	7
1.4.5	Індуктивні типи	8
1.4.6	Числення процесів	9
1.5	Дослідження систем типів	10
1.5.1	PTS системи	10
1.5.2	MLTT системи	10
1.5.3	HTS системи	10
1.5.4	Pi-числення	10
1.5.5	Числення алгебраїчних ефектів	10
1.5.6	Обробка векторних масивів	10
1.6	Структура роботи	11
1.7	Дослідження середовищ виконання	12
1.8	Продукти дослідження	13
1.8.1	Інтерпретатор O на Rust для L4	13
1.8.2	Коіндукція на Coq з екстрактом в OCaml	13
1.8.3	Екстракція в Erlang та O з OM	13

2	Система з однією аксіомою	15
2.1	Синтаксис	16
2.2	Всесвіти	17
2.3	Предикативні всесвіти	17
2.4	Імпредикативні всесвіти	18
2.5	Система з однією аксіомою	18
2.6	Ієрархії	19
2.7	Перевірка всесвітів	19
2.8	Перевірка П-типів	19
2.9	Перевірка змінних	19
2.10	Індекси де Брейна	19
2.11	Нормалізація	19
2.12	Підстановка	20
2.13	Рівність за визначенням	20
2.14	Перевірка типів	20
2.15	Екстракт в платформу Erlang/OTP	20
2.16	Індуктивні типи	22
2.17	Поліноміальні функтори	22
2.18	Кодування List	23
2.19	Нормальні форми	24
3	Інтерпретатор	25
4	Базова бібліотека	27
5	Система доведення теорем	29
	Бібліографія	30

# Глава 1

## Вступ

---

Присвячується Маші та Міші

---

### 1.1 Системна інженерія та верифікація

Протягом історії обчислювальної техніки було створено різні класи та способи обчислень, різні теорії та підходи до програмування таких систем, різні класи систем програмування. Зараз уже стало зрозумілим, що інженіринг систем які не піддаються до верифікації формальними методами не може бути застосований у галузях де вимоги до якості особливо підвищені, як то космонавтика, енергетика та фінанси.

Об'єктом дослідження данної роботи є системи верифікації програмного забезпечення та операційні системи які виконують обчислення в реальному часі, їх поєднання та побудова формальної системи для уніфікованого середовища, яке поєднує середовище виконання та систему верифікації у єдину систему мов. Розрізняють наступні класи систем верифікації (систем доведення теорем, або пруверів): 1) автоматизовані системи, які зосереджені на максимальній автоматизації процесу доведення (Z3, HOL, Coq); 2) системи програмування з конвертацією доведених програм в довільні мови (Idris, F\*, Coq); 3) логічні фреймворки зі спеціальними логічними мовами для верифікації інших мов (Twelf, TLA+, NuPRL).

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. Сама система доведення теорем є логікою. Імплементация мови програмування, яка релізує

логічну семантику здійснюється завдяки теорії типів. Формалізація методів відбувається завдяки теорії категорій, яка є абстрактною алгеброю функцій, метематичним інструментом для формалізації мов програмування та довільних математичних теорій які описуються логіками вищих порядків.

## 1.2 Історія систем верифікації

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 Принстоні. Ця мова була побудована на конструктивних примітивах, які виявилися компонентами індуктивних конструкцій які були структуровані пізніше і формалізовані за допомогою теорії категорій Вільяма Лавіра. Окрім LISP, нетипізоване лямбда числення маніфестується у такі мови як Erlang, JavaScript, Python. До цих пір нетипізоване лямбда числення є одною з мов у які робиться конвертація доведених програм (екстракція).

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

ML/LCF або метамова і логіка обчислювальних функцій був наступник крок до досягнення фундаментальної мови простору, тут вперше з'явилися алебраїчні типи даних у вигляді індуктивних типів, поліноміальних функторів або терміновані (well-founded) дерев. Роберт Мілнер, асистований Морісом та Н'юві розробив Метамову (ML), як інструмент для побудови прuverа LCF. LCF був основоположником у родині прuverів HOL88, HOL90, HOL98 та останньої версії на даний час HOL/Isabell. Пізніше були побудовані категорні моделі Татсоя Хагіно (CPL, Японія) то Робін Кокетом (Charity, Канада).

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

### 1.3 Методи верифікації

Можна виділити два 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. VST (CompCert, сертифікація C програм), NuPRL (Cornell University, розподілені системи, залежні типи), TLA+ (Microsoft Research, Леслі Лампорт), Twelf (для верифікації мов програмування).

Інший підхід можна назвати підходом вбудованих DSL. Усе моделювання відбувається в основній мові, а сертифіковані програми автоматично екстрагуються в довільні мови. Приклади таких систем: Coq побудована на мові OCaml від науково-дослідного інституту Франції INRIA; Agda побудовані на мові Haskell від шведського інституту технологій Чалмерс; Lean побудована на мові C++ від Microsoft Research та Університету Каргені-Мелона; Idris побудована на мові Haskell Едвіна Бреді з шотландського Університету ім. св. Андрія; F\* – окремий проект Microsoft Research.

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 верифікації.

## 1.4 Метематичне забезпечення

### 1.4.1 Інтуїціоністична теорія типів Мартіна-Льофа

Пер Мартін-Льоф в 1972 році запропонував  $\Pi$ ,  $\Sigma$  та  $\text{Id}$  у якості основних фундаментальних типів. З тих пір усі сучасні системи типів для перевірок побудовані наслідуючи цю модель. Було показано, що мова такої системи типів є внутрішнією мовою локальних декартово-замкнених категорій.  $\Pi$  та  $\Sigma$  кодують безпосередньо логічні квантори  $\forall$  та  $\exists$ , а тип  $\rightarrow$  є частковим випадком  $\Pi$ -типу, коли вираз  $B$  не залежить від  $x$ .

Аксіоми:

$$\frac{\Gamma x : A \vdash B : \text{Type} \quad \Gamma \vdash A : \text{Type}}{\Gamma \vdash \Pi(x : A) \rightarrow B(x) : \text{Type}} \quad (\Pi)$$

$$\frac{\Gamma x : A \vdash B : \text{Type} \quad \Gamma \vdash A : \text{Type}}{\Gamma \vdash \Sigma(x : A) \times B(x) : \text{Type}} \quad (\Sigma)$$

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

Теорема:

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



## 1.4.2 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 тотожним морфізмом (тотальною функцією — одиницею) який існує для кожного об'єкту (типу) категорії. Аксиоми формації об'єктів не приводяться та автопостулюються в нижніх аксіомах. Поки що тут буде визначатися тільки композиція морфізмів. Об'єкти  $A$  та  $B$  морфізма  $f : A \rightarrow B$  називаються домен та кодомен відповідно. Композиція є фундаментальною властивістю морфізмів.

Інтро аксіоми – асоціативність композиції та права і ліва композиції одиниці показують, що категорії є типизованими 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.

Аксиоми:

$$\begin{array}{c}
 \frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash g : B \rightarrow C}{\Gamma \vdash g \circ f : A \rightarrow C} \qquad \frac{}{\Gamma \vdash \text{id}_A : A \rightarrow A} \\
 \\
 \frac{\Gamma \vdash f : B \rightarrow A \quad \Gamma \vdash g : C \rightarrow B \quad \Gamma \vdash h : D \rightarrow C}{\Gamma \vdash (f \circ g) \circ h = f \circ (g \circ h) : D \rightarrow A} \qquad \frac{\Gamma \vdash f : A \rightarrow B}{\Gamma \vdash f \circ \text{id}_A = f : A \rightarrow B} \\
 \\
 \frac{}{\Gamma \vdash \text{id}_B \circ f = f : A \rightarrow B}
 \end{array}$$

### 1.4.3 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-повідомлення, а також існує  $\perp$  тип-термінал, та  $\top$  тип-котермінал. Термінальними типами зручно термінувати рекурсивні типи даних, такі як списки. Ми будемо розглядати тільки категорії які мають добутки та суми.

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

Аксиоми:

$$\begin{array}{c}
 \frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash g : A \rightarrow C \quad \Gamma \vdash B \times C}{\Gamma \vdash \langle f, g \rangle : A \rightarrow B \times C} \quad \frac{}{\Gamma \vdash \top} \\
 \\
 \frac{\Gamma x : A \times B}{\Gamma \vdash \pi_1 : A \times B \rightarrow A; \Gamma \vdash \pi_2 : A \times B \rightarrow B} \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash a \mid b : A + B} \\
 \\
 \frac{\Gamma x : A + B}{\Gamma \vdash \sigma_1 : A \rightarrow A + B; \Gamma \vdash \sigma_2 : B \rightarrow A + B} \\
 \\
 \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B} \quad \frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash g : A \rightarrow C \quad \Gamma \vdash B + C}{\Gamma \vdash [f, g] : A \rightarrow B + C} \\
 \\
 \frac{}{\Gamma \vdash \perp}
 \end{array}$$

Теорема:

$$\begin{aligned}
 (f \circ g) \circ h &= f \circ (g \circ h) \\
 f \circ \text{id} &= f \\
 \text{id} \circ f &= f \\
 \pi_1 \circ \langle f, g \rangle &= f
 \end{aligned}$$

$$\begin{aligned}
 \pi_2 \circ \langle f, g \rangle &= g \\
 \langle f \circ \pi_1, f \circ \pi_2 \rangle &= f \\
 \langle f, g \rangle \circ h &= \langle f \circ h, g \circ h \rangle \\
 \langle \pi_1, \pi_2 \rangle &= \text{id}
 \end{aligned}$$

## 1.4.4 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 типіох системах, таких як System F $\omega$ , та системах Haskell та Scala.

З категоріальні точки зору експоненти  $f : A^B$  є аналогами функціональних просторів  $f : B \rightarrow A$ . 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  $\Gamma$ ; та її елімінатора – операції аплікації функції до аргументу. Так визначається декартово-замкнена категорія. Визначається також рекурсивний механізм виклику функції з довільною кількістю аргументів.

Аксіоми:

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

$$\frac{\Gamma f : A \rightarrow B \quad \Gamma a : A}{\Gamma \vdash \text{apply } f \ a : (A \rightarrow B) \times A \rightarrow B}$$

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

Теорема:

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

Об'єкти :  $T \mid \perp \mid \rightarrow \mid \times \mid +$ Морфізми :  $\text{id} \mid f \circ g \mid [f, g] \mid \langle f, g \rangle \mid \text{apply} \mid \lambda \mid \text{curry}$

### 1.4.5 Індуктивні типи

Системи з залежними типами як верифікаційні математичні формальні моделі для доведення коректності. Система  $\Sigma$  та  $\Pi$  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 поліморфні індуктивні структури та коіндуктивні структури. В Lean теж відсутні коіндуктивні структури проте повністю реалізована теорія HoTT на нерекурсивних поліморфних структурах що об'єднує 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.

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

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

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

## 1.4.6 Числення процесів

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

$$\begin{array}{c}
 \frac{\Gamma \vdash E, \Sigma, X \quad \Gamma \vdash \text{action} : \Sigma \times X \rightarrow \Sigma \times X}{\Gamma \vdash \text{spawn action} : \pi_\Sigma} \\
 \\
 \frac{\Gamma \vdash \text{pid} : \pi_\Sigma \quad \Gamma \vdash \text{msg} : \Sigma}{\Gamma \vdash \text{join msg pid} : \Sigma \times \pi_\Sigma \xrightarrow{\bullet} \Sigma; \Gamma \vdash \text{send msg pid} : \Sigma \times \pi_\Sigma \rightarrow \Sigma} \\
 \\
 \frac{\Gamma \vdash L : A + B, R : X + Y \quad \Gamma \vdash M : A \rightarrow X, N : B \rightarrow Y}{\Gamma \vdash \text{receive } L M N : L \xrightarrow{\bullet} R}
 \end{array}$$

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

$$\begin{array}{ll}
 \oplus & : \quad \pi \parallel \pi \\
 \otimes & : \quad \pi | \pi
 \end{array}$$

## 1.5 Дослідження систем типів

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

В ході дослідження були виявлені основні типові системи, або мовні рівні

### 1.5.1 PTS системи

### 1.5.2 MLTT системи

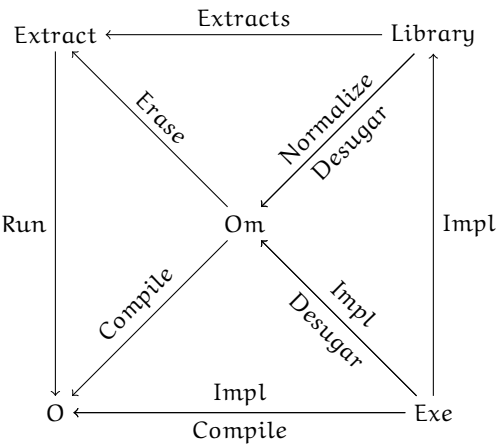
### 1.5.3 HTS системи

### 1.5.4 Рі-числення

### 1.5.5 Числення алгебраїчних ефектів

### 1.5.6 Обробка векторних масивів

1.6 Структура роботи



$\text{Exe} \xrightarrow{\text{Impl}} \text{Om}$	— На прувері пишеться ядро
$\text{Exe} \xrightarrow{\text{Impl}} \text{O}$	— На прувері пишеться інтерпретатор
$\text{Exe} \xrightarrow{\text{Impl}} \text{Exe}$	— Прувер написаний сам на собі
$\text{Exe} \xrightarrow{\text{Impl}} \text{Library}$	— На прувері пишеться базова бібліотека
$\text{Exe} \xrightarrow{\text{Desugar}} \text{Om}$	— Сам прувер є розширенням ядра
$\text{Library} \xrightarrow{\text{Desugar}} \text{Om}$	— Базова бібліотека конвертується в код ядра
$\text{Library} \xrightarrow{\text{Normalize}} \text{Om}$	— Повна нормалізація базової бібліотеки
$\text{Om} \xrightarrow{\text{Erase}} \text{Extract}$	— Видаляється інформація про типи, детипізація
$\text{Extract} \xrightarrow{\text{Run}} \text{O}$	— Запуск на інтерпретаторі

## 1.7 Дослідження середовищ виконання

Усі середовища виконання можна умовно розділити на два класи:

1) інтерпретатори нетипізованого або просто типізованого (рідше з більш потужними системами типів), лямбда числення з можливими JIT оптимізаціями; та 2) безпосередня генерація інструкцій процесора і лінування цієї програми з середовищем виконання що забезпечує планування ресурсів (в цій області переважно використовується System F типізація).

До першого класу можна віднести такі віртуальні 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 середовища виконання.

Найбільш цікаві цільові платформи для виконання програм 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 заблоковану семантику пі-калькула) та Pony (тому, що семантика його пі-калькула побудована на імутабельних чергах та CAS курсорах). У цій роботі ми зосередимося на дослідженні трьох підходів та побудові трьох прототипів.



## 1.8 Продукти дослідження

### 1.8.1 Інтерпретатор O на Rust для L4

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

### 1.8.2 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.

### 1.8.3 Екстракція в Erlang та O з OM

Третій прототип – побудова тайпчекера та екстрактора у мову Erlang та O. Ця робота представлена у вигляді PTS тайпчекера OM, який виступає у ролі проміжної мови для повної нормалізації лямбда термів. В роботі використане нерекурсивне кодування індуктивних типів та продемонстрована теж бескінечна тотальна програма у якості способу лінування з підсистемою вводу-виводу віртуальної машини Erlang.

### Idris та PureScript разом з Erlang

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



## Система з однією аксіомою

---

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 Льюфа існує чотири конструкції: формація, інтро, елімінатор та редуктор.

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

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

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

## 2.1 Синтаксис

Om syntax is compatible with  $\lambda C$  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 >
0  ::= U
      | I | ( 0 ) | 0 0 | 0  $\rightarrow$  0
      |  $\lambda$  ( I : 0 )  $\rightarrow$  0
      |  $\forall$  ( I : 0 )  $\rightarrow$  0
```

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)
```

## 2.2 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.

$U_0 : U_1 : U_2 : U_3 : \dots$

$U_0$  --- propositions  
 $U_1$  --- values and sets  
 $U_2$  --- types  
 $U_3$  --- sorts

(S)

$$\frac{o : \text{Nat}}{\text{Type}_o}$$

## 2.3 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):

$S \quad (n : \text{nat}) = U \ n$   
 $A_1 \ (n \ m : \text{nat}) = U \ n : U \ m \text{ where } m > n \quad - \text{cumulative}$   
 $R_1 \ (m \ n : \text{nat}) = U \ m \rightarrow U \ n : U \ (\max \ m \ n) \quad - \text{predicative}$

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

(A<sub>1</sub>)

$$\frac{i : \text{Nat}, j : \text{Nat}, i < j}{\text{Type}_i : \text{Type}_j}$$

(R<sub>1</sub>)

$$\frac{i : \text{Nat}, j : \text{Nat}}{\text{Type}_i \rightarrow \text{Type}_j : \text{Type}_{\max(i,j)}}$$

## 2.4 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.

$A_2 (n : \text{nat}) = U n : U (n + 1) \quad - \text{non-cumulative}$   
 $R_2 (m n : \text{nat}) = U m \rightarrow U n : U n \quad - \text{impredicative}$

$$\frac{i : \text{Nat}}{\text{Type}_i : \text{Type}_{i+1}} \quad (A_2)$$

$$\frac{i : \text{Nat}, \quad j : \text{Nat}}{\text{Type}_i \rightarrow \text{Type}_j : \text{Type}_j} \quad (R_2)$$

## 2.5 Система з однією аксіомою

This language 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.

$\forall (x : A) \rightarrow B \quad x : \text{Type}$   
 $\lambda (x : A) \rightarrow b : B \quad x$   
 $f \quad a : B \quad [a/x]$   
 $(\lambda (x : A) \rightarrow b) \quad a = b[a/x] : B[a/x]$

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

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

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

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

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)
```

## 2.6 Ієрархії

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

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

## 2.7 Перевірка всесвітів

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

## 2.8 Перевірка Π-типів

```
fun ((:∀,), (I,0)) → true
fun T              → (:error, (:∀, T))
```

## 2.9 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)))
```

## 2.10 Індокси де Брейна

```
sh (:var, (N, I)), (N, P) when I >= P → (:var, (N, I+1))
sh ((:∀, (N, 0)), (I, 0)), (N, P)     → ((:∀, (N, 0)), sh I N P, sh 0 N P+1)
sh ((:λ, (N, 0)), (I, 0)), (N, P)     → ((:λ, (N, 0)), sh I N P, sh 0 N P+1)
sh (Q, (L, R), N, P)                  → (Q, sh L N P, sh R N P)
sh (T, N, P)                          → T
```

## 2.11 Нормалізація

```
norm :none          → :none
norm :any           → :any
norm (:app, (F, A)) → case norm F of
                        ((:λ, (N, 0)), (I, 0)) → norm (sub 0 N A)
                        NF → (:app, (NF, norm A)) end
norm (:remote, N)   → cache (norm N [])
norm (→, (I, 0))   → ((:∀, ("_", 0)), (norm I, norm 0))
norm ((:∀, (N, 0)), (I, 0)) → ((:∀, (N, 0)), (norm I, norm 0))
norm ((:λ, (N, 0)), (I, 0)) → ((:λ, (N, 0)), (norm I, norm 0))
norm T              → T
```

## 2.12 Підстановка

```

sub Term Name Value          → sub Term Name Value 0
sub (:→, (I,0)) N V L → (:→, sub I N V L, sub 0 N V L);
sub ((:∀,(N,0)),(I,0)) N V L → ((:∀,(N,0)),sub I N V L,sub 0 N(sh V N 0)L+1)
sub ((:∀,(F,X)),(I,0)) N V L → ((:∀,(F,X)),sub I N V L,sub 0 N(sh V F 0)L)
sub ((:λ,(N,0)),(I,0)) N V L → ((:λ,(N,0)),sub I N V L,sub 0 N(sh V N 0)L+1)
sub ((:λ,(F,X)),(I,0)) N V L → ((:λ,(F,X)),sub I N V L,sub 0 N(sh V F 0)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.

```

## 2.13 Рівність за визначенням

```

eq ((:∀,("_",0)), 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 ((:∀,(N1,0)),(I1,01)) ((:∀,(N2,0)),(I2,02)) →
  let true = eq I1 I2 in eq 01 (sub (sh 02 N1 0) N2 (:var,(N1,0)) 0)
eq ((:λ,(N1,0)),(I1,01)) ((:λ,(N2,0)),(I2,02)) →
  let true = eq I1 I2 in eq 01 (sub (sh 02 N1 0) N2 (:var,(N1,0)) 0)
eq (A,B) → (:error,(:eq,A,B))

```

## 2.14 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,0))      D → (:*,h(star(type I D)),star(type 0 D))
type ((:∀,(N,0)),(I,0)) D → (:*,h(star(type I D)),star(type 0 [(N,norm I)|D]))
type ((:λ,(N,0)),(I,0)) D → let star (type I D),
  NI = norm I in ((:∀,(N,0)),(NI,type(0,[(N,NI)|D])))
type (:a,(F,A))      D → let T = type(F,D),
  true = fun T,
  ((:∀,(N,0)),(I,0)) = T,
  Q = type A D,
  true = eq I Q in norm (subst 0 N A)

```

## 2.15 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 compiles 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

```

## 2.16 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 without 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.

## 2.17 Поліноміальні функтори

Least fixed point trees are called well-founded trees and encode polynomial functors.

Natural Numbers:  $\mu X \rightarrow 1 + X$   
 List A:  $\mu X \rightarrow 1 + A \times X$   
 Lambda calculus:  $\mu X \rightarrow 1 + X \times X + X$   
 Stream:  $\nu X \rightarrow A \times X$   
 Potentially Infinite List A:  $\nu X \rightarrow 1 + A \times X$   
 Finite Tree:  $\mu X \rightarrow \mu Y \rightarrow 1 + X \times Y = \mu X = \text{List } X$

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 identity signature all the constructor components of (co)inductive type.

## 2.18 Кодування List

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

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

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

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

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

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

```
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))
```

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

## 2.19 Нормальні форми

## Lists/Map

$$\begin{aligned}
& \lambda (a: *) \rightarrow \lambda (b: *) \rightarrow \lambda (f: a \rightarrow b) \rightarrow \lambda (xs: \forall (List: *) \rightarrow \forall (Cons: \forall \\
& (\text{head}: a) \rightarrow \forall (\text{tail}: List) \rightarrow List) \rightarrow \forall (Nil: List) \rightarrow List) \rightarrow xs (\forall (List: \\
& *) \rightarrow \forall (Cons: \forall (\text{head}: b) \rightarrow \forall (\text{tail}: List) \rightarrow List) \rightarrow \forall (Nil: List) \rightarrow \\
& List) (\lambda (\text{head}: a) \rightarrow \lambda (\text{tail}: \forall (List: *) \rightarrow \forall (Cons: \forall (\text{head}: b) \rightarrow \forall \\
& (\text{tail}: List) \rightarrow List) \rightarrow \forall (Nil: List) \rightarrow List) \rightarrow \lambda (List: *) \rightarrow \lambda (Cons: \forall \\
& (\text{head}: b) \rightarrow \forall (\text{tail}: List) \rightarrow List) \rightarrow \lambda (Nil: List) \rightarrow Cons (f \text{ head}) (\text{tail} \\
& List \text{ Cons Nil})) (\lambda (List: *) \rightarrow \lambda (Cons: \forall (\text{head}: b) \rightarrow \forall (\text{tail}: List) \rightarrow \\
& List) \rightarrow \lambda (Nil: List) \rightarrow Nil)
\end{aligned}$$

## Глава 3

# Інтерпретатор

---



## 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)
  (putLine: 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

# Система доведення теорем

---

# 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 Families 1997
- [12] B.Jacobs (Co)Algebras and (Co)Induction 1997
- [13] V.Vene Categorical programming with (co)inductive types 2000
- [14] H.Geuevers 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