Дисертація

Система верифікації програмного забезпечення

Максим Сохацький, Павло Маслянко

Зміст

Глава 1

Вступ

1.1 Системна інженерія та верифікація

Протягом історії обчислювальної техніки було створено різні класи та способи обчислень, різні тоеорії 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 яки виконують обчислення в реальному часі, їх поєднання та побудова формальної системи для унифікованого середовища, яке поєднує середовище виконання та систему верифікації у єдину систему мов.

Предметом дослідження такої системи мов є теорія типів, яка вивчає обчислювальні властивості мов. Теорія типів виділилася в окрему науку Мартіном Льофом як запит на ваканте місце у трикутнику теорій, які відповідають ізоморфізму Каррі-Говарда (Логіки, Мови, Категорії). Інші дві це: теорія категорій та логіка вищих порядків. Сама система доведення теорем є логікою, або аспектом логіки у трикутнику. Імплементація мови програмування, яка релізує логічну семантику здійснюється завдяки теорії типів. Формалізація методів відбувається завдяки теорії категорій, яка є абстрактною алгеброю фунцій, 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 Прінстоні. Ця мова була побудована на таких примітивах, як: cons, nil, eq, atom, car, cdr, lambda, apply and id. Направді це уривки індуктивних конструкцій які були структуровані пізніше і формалізовані за допомогою теорії категорій Вільяма Лавіра. До цих пір нетипізоване лямбла числення є одною з мов у які робиться екстракт з сучасних пруверів. Окрім LISP, нетипізоване лямбда числення маніфестується у такі мови як Erlang, JavaScript, Python.

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.

ML/LCF або метамова і логіка обчисльювальних функцій був наступник крок до осягнення фундаментальної мови простору, тут впреше з'явилися алебраїчні типи даних у вигляді індуктивних типів, поліноміальних функторів або терміновані (well-founded) дерев. Пізніше були побувані категорні моделі Татсоя Хагіно (CPL) то Крокетом (Charity). 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.

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).

1.3 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 Сі-програм), NuPRL (Cornell University, розподілені системи, залежні типи), TLA+ (Microsoft Reseach, Леслі Лампорт), Twelf (для верифікації мов програмування).

Інший підхід можна назвати підходом вбудованих DSL. Усе моделювання відбувається в основній мові, а сертифіковані програми автоматично екстрактяться 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 Універсистету Каргені-Мелона; Idris подудована 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 фьюжином фундаментальної математики та інженерних систем з формальними методами верицікації. Методи цього дослідження є суто теоретичними.

1.4 Метематичне забезпечення

1.4.1 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. Аксіома формації морфізмів буде даватися як введеня експоненти після визначення декартового добутку. Поки що тут буде визначатися тільки композиція морфізмів. Об'єкти A та B морфізма $f : A \to B$ називаються домен та кодомен відповідно.

Інтро аксіоми – асоціативність композиції та права і ліва композиції одниці 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.

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

$$\frac{}{\Gamma \vdash id_A : A \to A}$$

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

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \circ id_A = f : A \to B}$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash id_B \circ f = f : A \to B}$$

Композиція показує можливість звязувати область значень попереднього обчислення (кодомен) та область визначення наступного обчислення (домен). Композиція є фундаментальною властивістю морфізмів.

1. $A : *$
2. $A : * \, , \; B : * \implies f : A \to B$
3. $f : B \to C \, , \; g : A \to B \implies f \circ g : A \to C$
4. $(f \circ g) \circ h = f \circ (g \circ h)$
5. $A \implies id : A \to A$
6. $f \circ id = f$
7. $id \circ f = f$

1.4.2 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 $\perp$ 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 обернені 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 $\pi \circ \sigma = \sigma \circ \pi = id$.

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

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

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

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

$$\frac{}{\Gamma \vdash \ \perp}$$

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

Також додамо тут аксіому множення морфізмів, яка випливає з визначення добутка, яка необхідна для забезпечення аплікативного програмування.

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

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

1.4.3 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, таких як System Fω, системах Haskell та Scala.

Щоб пояснити функції з категоріальні точки зору потрібно пояснити категоріальні експоненти $f : A^B$, які є аналогами фукціональних просторів $f : A \to B$. 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.

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

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

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

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

Об'єкти : $\perp \mid\ \to\ \mid \times$

Морфізми : $\mathrm{id} \mid f \circ g \mid \langle f, g \rangle \mid \mathrm{apply} \mid \lambda \mid \mathrm{curry}$

1.4.4 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 кортежа та є морфізмом-одиницею типу свого стану.

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

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

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

Алгебра процесів визначає базові операції мультиплексування двох чи декількох протоколів в рамках одного процесу (добуток), а також паралельного та повністю ізольованого запуску включно зі стеком та областю памяті (сума) на віртуальній машині.

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

1.4.5 Інтуітіоністична теорія типів Мартіна Льофа

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 використувують 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. В Lean також підтримуються типи класів.

1.4.6 Логіка та квантори

Модель логік вищого порядку та квантори ∀ та ∃ які теж виража-ються як конструкції типів:

$$\frac{\Gamma\, x : A \vdash B \qquad \Gamma\ \vdash A}{\Gamma\ \vdash \Pi(x : A)B} \quad \frac{\Gamma\ \vdash a : A \qquad \Gamma\, x : A \vdash B \qquad \Gamma\, b : B(x = a)}{\Gamma\ \vdash (a, b) : \Pi(x : A)B}$$

$$\frac{\Gamma\, x : A \vdash B \qquad \Gamma\ \vdash A}{\Gamma\ \vdash \Sigma(x : A)B} \quad \frac{\Gamma\ \vdash a : A \qquad \Gamma\, x : A \vdash B \qquad \Gamma\, b : B(x = a)}{\Gamma\ \vdash (a, b) : \Sigma(x : A)B}$$

$$\frac{\Gamma\ \vdash x : A \qquad \Gamma\ \vdash x' : A}{\Gamma\ \vdash \mathrm{Id}_A(x, x')}$$

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

1.5 Дослідження середовищ виконання

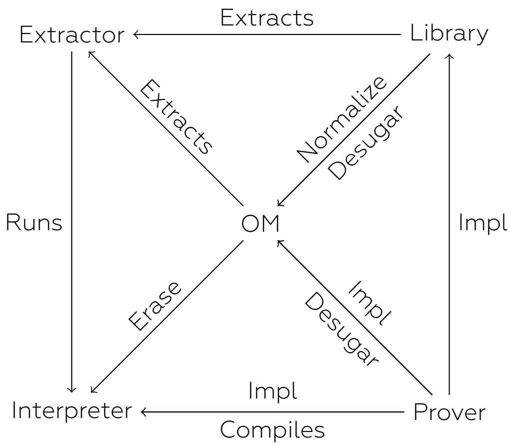
1.5.1 Agda та M-Alonso

1.5.2 Coq та coq.io

1.5.3 Rust

1.5.4 Erlang

1.6 Архітектура компонентів системи



Глава 2

Система з однією аксіомою

2.1 Мова зі строгою нормалізацією

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 $Axioms$ 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:

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

An intermediate Om language is based on Henk [6] languages described first by Erik Meijer and Simon Peyton Jones in 1997. 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: $apply$ and $curry$, infinity number of universes, and one computation rule called β-reduction. The design of Om language resemble Henk and Morte both design and implementation. This language indended to be small, concise, easy provable and able to produce verifiable peace of code that can be distributed over the networks, compiled at target with safe trusted linkage.

2.1.1 BNF

Om syntax is compatible with λC 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:

```
Inductive OM := Star: nat → OM
              | Var: name → OM
              | App: OM → OM → OM
              | Lambda: name → OM → OM → OM
              | Arrow: OM → OM → OM
              | Pi: name → OM → OM → OM.
```

2.1.2 Universes

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
```

(S)
$$\frac{o : Nat}{Type_o}$$

2.1.3 Predicative Universes

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.

(A_1)
$$\frac{i : Nat, j : Nat, i < j}{Type_i : Type_j}$$

(R_1)
$$\frac{i : Nat, j : Nat}{Type_i \to Type_j : Type_{max(i,j)}}$$

2.1.4 Impredicative Universes

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
```

$$\frac{i : \mathrm{Nat}}{\mathrm{Type}_i : \mathrm{Type}_{i+1}} \tag{A_2}$$

$$\frac{i : \mathrm{Nat}, \quad j : \mathrm{Nat}}{\mathrm{Type}_i \to \mathrm{Type}_j : \mathrm{Type}_j} \tag{R_2}$$

2.1.5 Single Axiom Language

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]
```

$$\frac{x : A \vdash B : \mathrm{Type}}{\Pi\ (x : A) \to B : \mathrm{Type}} \tag{Π-formation}$$

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

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

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

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)
```

2.1.6 Hierarchy

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

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

2.1.7 Universes

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

2.1.8 Functions

```
func ((:forall,),(I,O)) → true
func T                  → (:error,(:forall,T))
```

2.1.9 Variables

```
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)))
```

2.1.10 Shift

```
sh (:var,(N,I)),N,P) when I>=P  → (var,(N,I+1))
sh ((:forall,(N,O)),(I,O)),N,P) → ((:forall,(N,O)),sh I N P,sh O N P+1)
sh ((:lambda,(N,O)),(I,O)),N,P) → ((:lambda,(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
```

2.1.11 Substitution

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

2.1.12 Normalization

```
norm :none                      → :none
norm :any                       → :any
norm (:app,(F,A))               → case norm F of
                                  ((:lambda,(N,O)),(I,O)) → norm (subst O N A)
                                                       NF → (:app,(NF,norm A)) end
norm (:remote,N)                → cache (norm N [])
norm (:arrow,         (I,O))  → ((:forall,("_",O)),(norm I,norm O))
norm ((:forall,(N,O)),(I,O))  → ((:forall,(N,O)),  (norm I,norm O))
norm ((:lambda,(N,O)),(I,O))  → ((:lambda,(N,O)),  (norm I,norm O))
norm T                          → T
```

2.1.13 Definitional Equality

```
eq ((:forall,("_",0)), X) (:arrow,Y)     → eq X Y
eq (:app,(F1,A1))         (:app,(F2,A2)) → let true = eq F1 F2 in eq A1 A2
eq (:star,N)              (:star,N)      → true
eq (:var,(N,I))           (:var,(N,I))   → true
eq (:remote,N)            (:remote,N)    → true
eq ((:farall,(N1,0)),(I1,O1))
   ((:forall,(N2,0)),(I2,O2)) →
   let true = eq I1 I2 in eq O1 (subst (shift O2 N1 0) N2 (:var,(N1,0)) 0)
eq ((:lambda,(N1,0)),(I1,O1))
   ((:lambda,(N2,0)),(I2,O2)) →
   let true = eq I1 I2 in eq O1 (subst (shift O2 N1 0) N2 (:var,(N1,0)) 0)
eq (A,B)                         → (:error,(:eq,A,B))
```

2.1.14 Type Checker

```
type (:star,N)                _ → (:star,N+1)
type (:var,(N,I))             D → let true = var N D in keyget N D I
type (:remote,N)              D → cache type N D
type (:arrow,(I,O))           D → (:star,h(star(type I D)),star(type O D))
type ((:forall,(N,0)),(I,O)) D → (:star,h(star(type I D)),star(type O [(N,norm I)|D]))
type ((:lambda,(N,0)),(I,O)) D → let star (type I D),
                                     NI = norm I in ((:forall,(N,0)),(NI,type(O,[(N,NI)|D])))
type (:app,(F,A))             D → let T = type(F,D),
                                     true = func T,
                                     ((:forall,(N,0)),(I,O)) = T,
                                     Q = type A D,
                                     true = eq I Q in norm (subst O N A)
```

2.1.15 Target Erlang VM and LLVM platforms

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

2.2 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.

2.2.1 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

2.2.2 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
```

2.2.3 Inductive Types

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.

2.2.4 Polynomial Functors

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: $\nu\, X \rightarrow A \times X$
Potentialy Infinite List A: $\nu\, X \rightarrow 1 + A \times X$
Finite Tree: $\mu\, X \rightarrow \mu\, Y \rightarrow 1 + X \times Y = \mu\, X = \mathrm{List}\ X$

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

2.2.5 Lists

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

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

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

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

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

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

```
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))
```

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

2.2.6 Normal Forms

Lists/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)

2.2.7 Prelude Base Library

```
  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 3

Інтерпретатор

Глава 4

Базова бібліотека

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 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