Система доведення теорем з однією аксіомою

Максим Сохацький

[1] Національний технічний університет України
ім. Ігоря Сікорського

10 грудня 2019 р.

Анотація

Ця стаття презентує диайн мови програмування $\mathbf{PTS}^\infty$, імплiментації її типового верифікатора, а також екстрактор байткоду для віртуальної машини Erlang від Ericsson. $\mathbf{PTS}^\infty$ — це мова проміжного рівня заснована на так званій чистій системі типів, або системі типів з однією аксіомою та зліченною кількістью всесвітів (консистента теорія залежних типів). Ця мова програмування дає зручну мову проміжного рівня для застосунку у додадках з підвищеними вимогами до математичної верифікації. Типовий верифікатор побудований за базі MLTT принципів та конфігурується правилами для предикативної та імпредактивної ієрархії всесвітів. Синтаксис цієї мови програмування сумісний з базовим синтаксистом мови Morte, та підтримує її базову бібліотеку, а також додає поняття нескінченної кількості всесвітів. Дуже базова бібліотека прилюдії з рекрурсивним та корекурсивним вводом-вивдом додаються як частина цієї роботи. Це дає змогу застосовувати основи математичної верифікації до нескінченних або довгоживучих процесів.

Ми коротко опишемо мову верхнього рівня, та мову проміжного ядра для описаного в статті типового верифікатора, та покажемо місце цієї мови у конценптуальній системі доведення теорем, яка передбачає поєднання: 1) оптимального лямбда обчислювача; 2) мову з однією аксіомою; 3) MLTT мову; 4) мову з гомотопічними типами та інтервалом.

Зміст

1 Вступ

IEEE[1] стандарт та регуляторні документи ESA[2] визначають набір інструментів та підходів для процесу валідації та верифікації програмногоо забезпечення Найбіль розширені техніки передбачають застосування математичної логіки та теорії доведення теорем для формулювання виробничих задач у математичній формі для формальної перевірки коректності таких програм на всії області визначення функції з доведення властивостей цих функцій.

Ера верифікованих теорем, типових верифікатори та систем доведення теорем бере свій початок з доводжувача теорем AUTOMATH та теорії типів Мартіна-Льофа. Станом на сьогодні ми маємо такі потужні системи як Coq, Agda, Leam, F* які базуються на CoC (Calculus of Constructions, Coquand) та CiC (Calculus of Inductive Constructions, Paulin-Mohring). Подальший розвиток систематизації призвів до лямбда кубу та чистим системам з одією аксіомою, як узагальнююочому визначенню ситсем типу CoC, AUT-68, ECC, Henk, Morte, PTS$^\infty$. Головна мотивація систем з однією аксіомою — це простота імплементації та простота формальної імпленетації нормалізатора термів, як термінального обчислення. За допомогою формальної мови та типового верифікатора ми можемо передавати програми по відкритих каналах які задовільняють формальним типовим специфікацям та складним теоремам як властивостями цих об'єктів. У якості областей застосування тут можно виділити наступні сфери: 1) мови смарт-контрактів; 2) сертифіковані DSL; 3) платіжні системи, тощо.

1.1 Екстракція математично-доведених програм

Завдяки ізоморфізму Каррі-Ламбека-Ховарда — відповідності всередині теорії типів Мартіна-Льофа [13] між доведеннями, моделями та програмами, де типи, сигнатури та категорії є просторами (мовами) які містять у собі точки (програми), можемо трактувати терми як программи для обчислення певного результату, і природа цього обчислення може буде повністю позбавлена типізації, що дає змогу виконувати такі доведення на практивно довільному інтерпретаторі, так як майже всі так чи інакше реалізують модель лямбда-числення, тут маються на увазі мови JavaScript, Erlang, PyPy, LuaJIT, K. Також можна будувати екстрактори з інтераналізацією в C++, Rust. Ця робота головним чином презентує екстрактор в байт-код віртуальної машини Erlang, як модель простого нетипизованого лямбда числення, накшталт LISP або Smalltalk.

1.2 Системна архітектура

PTS$^\infty$ як мова програмування реалізую чисту систему типів, але зі зліченною кількістю всесвітів. Ця система типів формує основне мовне ядро си-

[1]IEEE Std 1012-2016 — V&V Software verification and validation
[2]ESA PSS-05-10 1-1 1995 – Guide to software verification and validation

стеми доведення теорем, усі інші системи типів містять чисту систему типів як підкатегорію у свому спектрі, та є її нащадками. З точки зору наслідкових зв'язків чиста система типів є основою усіх системах типів побудованих на розшаруваннях, П-типах, а також систем здатних до доведення теорем.

Поверх цієї базової ситеми типів виділяються інші системи типів які можна звести теж до одно-аксіоматичних систем з чітко-вираженим кодуваннями ізоморфізмів. До концептуальної моделі системи доведення теорем включатимемо наступні мовні ядра: 1) Мова з індуктивними типами для доведення у стилі математичної індукції; 2) Гомотопічне ядро з відкритим інтервалом для доведення у кубічному стилі; 3) Числення отоків як базис для тензорного числення (Futhark); 4) Числення процесів як базис для лінійних типів, коіндуктивного моделювання та середовища виконання. Незважаючи на те, що з усіх цих мовних рівней існують функтори в систему з однією аксіомою, ці мовні розширення програмуються як окремі плагіни функтори які погружаються у головний цикл типового верифікатора разом зі своїми правилами. Це дозволяє пришвидшити виконання нормалізації термів у процесі типової верифікації.

Однак не всі вищі мови можуть бути розкладені в базисі PTS. Як було показано Geuvers [9] ми не можемо побудувати принцип індукції всередині чистої PTS, ми повинні послабити до залучення оператора нерухомої точки принаймі для типової специфікації самого індуктивного рекурсивного типу. Також ми не можемо побудувати елімінатор рівності та функціональну екстенсіональність. Але незважаючи на це PTS це потужна система яка відразу дає змогу генерувати сертифіковані програми з доведених теорем. Властивосі можуть доводитися у інших мовах концептуальної моделі системи доведення теорем. Прошарок PTS більшим чином пов'язаний з мовою цільового інтерпретатору.

Як подальший розвиток цього проекту ми бачимо долучення індуктивної ситсеми типів та гомотопічної системи типів.

1.3 Місце серед інших мов

Продукт який представлено у статті виконаний для телекомунікаційної платформи Erlang/OTP від Ericsson, що дало змогу використовувати доведені програми на цій віртуальній машині. Цей додаток експонує наступні сервісу у середовищі Erlang: 1) типова верифікація; 2) нормалізація; 3) екстракція. Усі частини системи $\mathbf{PTS}^\infty$ написані на мові Erlang та виключно для системи Erlang.

- Рівень 0 — сертифікований векторизований інтерпретатор
- **Рівень 1 — консистентна система з однією аксіомою**
- Рівень 2 — вища мова для доведення теорем та перевірки властивостей

Табл. 1: Список мов, досліджених у якості цільової платформи для екстракції

Target	Class	Intermediate	Theory
C++	компілятор/native	HNC	System F
Rust	компілятор/native	HNC	System F
JVM	інтерпретатор/native	Java	F-sub[3]
JVM	інтерпретатор/native	Scala	System F-omega
GHC Core	компілятор/native	Haskell	System D
GHC Core	компілятор/native	Morte	CoC
Haskell	компілятор/native	Coq	CiC
OCaml	компілятор/native	Coq	CiC
BEAM	**інтерпретатор**	**Om**	**PTS$^\infty$**
O	інтерпретатор	Om	PTS$^\infty$
K	інтерпретатор	Q	Applicative
PyPy	інтерпретатор/native	N/A	ULC
LuaJIT	інтерпретатор/native	N/A	ULC
JavaScript	інтерпретатор/native	PureScript	System F

2 Консистентна мова проміжного рівня

Мова **PTS$^\infty$** є мовою з залежними типами, лямбда численням, розширенням CoC зі зліченною кількістю всесвітів для забезпечення консистентності. При цьому підтримуються два режима: предикативний та імпредикативний. В мові немає аксіоми нерухомої точки, що унеможливлює неконтрольовану рекурсію та парадокси в системі типів, а сама система насолоджується властивостями сильної нормалізації термів. Усі терми **Axioms** в такій системі відповідають своєму рангу всередині вкладеної послідовності всесвітів **Sorts**, а складність залежного терму залежить від максимальної складності самого терму та терму від якого він залежить, ця складність визначається правилами **Rules**. Систему всесвітів можна описати за допомогою SAR нотації Барендрехта [2]:

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

Синтаксис мови **PTS$^\infty$** базується на синтаксису мови Henk вперше описаний Еріком Мейером та Саймоном Пейтоном Джонсом в 1997 [12]. Пізніше, в 2015 з'явилася Haskell імплементація мови Henk — Morte, яка теж використовувала кодування Бома-Берардуччі для індуктивних та нерекурсивних лямбда термів. Ця мова базується виключно на П-типі, його інтро та елімінатор, зліченним всесвітам, бета-редукції та ета-експансії. **PTS$^\infty$** наслідує мови Henk та Morte як з точки зору дизайна, так і з точки зору імплементації, однак у нас більш оптимізовані індекси де Брейна, які застосовуются лише до одноіменних ідентифікаторів. Сама мова створена в дусі мінамілзма, та не залежить від зовнішніх бібліотек, у тому числі не залежить від лексичних аналізаторів та генетарів парсерів.

2.1 БНФ та Синтаксичне Дерево

Мова **PTS**$^{\infty}$ сумісна з системою типів CoC, яка реалізована в мовах Morte та Henk, однак містить ключове розширення — всесвіти можуть індексуватися натуральним числом **Nat**. Традиційно наводимо синтаксис мови у формі Бакуса-Наура, еквіваленте синтаксичне дерево наведено справа.

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

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

2.2 Всесвіти

Так як **PTS**$^{\infty}$ підтримує нескінченну зліченну кількість всесвітів, її метатеоричний індуктивний тип повинен містити натуральні числа. Самі всесвіти вбодовуються один в одного.

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

Де U_0 — всесвіт висловлювань, U_1 — всесвіт множин, U_2 — всесвіт типів, U_3 — всесвіт видів, тощо.

$$\frac{}{Nat} \tag{I}$$

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

Ви можете перевірити чи терм є всесвітом за допомогою наступної функції. Якщо аргумент не є всесвітом функцію верне помилку $\{error, \_\}$.

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

2.3 Предикативні всесвіти

Усі терми повинні відповідати рангу **Axioms** всередині послідовності всесвітів **Sorts**, та складність залежних термів **Rules** не повинна перевищувати складність бази та функтору, за допомогою якого породжений терм. Слід зауважити та наголосити, що предикативна ієрархія всесвітів не сумісна з

Чорч-кодуванням лямбда числення, у цьому можна переконатися спробувавши скомпілювати базову бібліотеку в режимі предикативних всесвітів.

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

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

2.4 Impredicative Universes

Як відомо, всесвіт висловлювань, або простір з одним населеним типом, можливо додати лише вниз ієрархії для забезпечення консистентності. Однак відкритим залишається питання нескінченної імпредикативної ієрархії.

$$\frac{i : Nat}{Type_i : Type_{i+1}} \tag{A_2}$$

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

2.5 Переклюлчення ієрархій

Імпредикативна версію функція визначення ієрархії **h** повертає цільовий всесвіт B розшарування B над A. Предикативна функція у свою чергу повертає максимальну потужність всесвітів B та A.

```
dep A B :impredicative → B
    A B :predicative   → max A B

h A B → dep A B :impredicative
```

2.6 Contexts

Контексти моделюють змінні в процессі розв'язання рівнянь типової сепцифікації. Контексти можна моделювати списками, векторами (Воєводський), та вкладеними сігмами (Канонічний спосіб). Правило елімінації тут не надається так як контекст в нашій імплементації повністю знищується після типової верифікації.

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

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

$$\frac{A : Type_i, \quad x : A, \quad \Gamma : Context}{(x : A) \ \vdash \ \Gamma : Context} \tag{Ctx-intro$_2$}$$

2.7 Система з однією аксіомою

Наша мова входить до класу чистих мов або мов з одним типом, так як всі правила виводу є компонентами кодування ізоморфізмом та виводяться з сигнатури самого П-типу. Єдині правила обчислення — бета-редукція та обернене до нього спряжене правило ета-експансії, їх зв'язок поєднується у двох додаткових правилах рівняннях лівої та правої одиничної композиції.

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

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

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

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

$$\frac{\pi_1 : A \quad u : A \vdash \pi_2 : B}{[\pi_1/u]\ \pi_2 : B} \tag{subst}$$

Теореми про PTS можуть бути вбудовані в саму PTS як логічний фреймворк для П-типів. Наведемо приклад в сиснтаксисі вищої мови.

```
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)
```

Терми навмисно наведені без доведень, так як можуть бути взяті з різних джерел [2]. Обчислювальна семантика бета та ета правил наведені у вищій мові як правила **Path**-типа. В реальній імплементації синтаксичне дерево мови **PTS**$^\infty$ розширене спеціальною вершиною для імпорта складних термів в текст программ на етапі завантаження термів з довіреного джерела. Ми також забороняємо рекурсію по цьому виду вершин.

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

Наш типовий верифікатор значно відрізняється від канонічного прикладу наведеним Террі Коканом [5], та складається з наступних примітивів: **Substitution**, variable **Shifting**, term **Normalization**, **Equality** за визначенням та власне **Type Checker**.

2.8 Верифікатор

В чистих системах ми повинні бути акуратними до рекурсії, тому як ми вже сказали ми забороняємо її відносно **:remote** вершин синтаксичного дерева. Йдеться мова про конструкції виду **#List/Cons or #List/map** де використовуються шляхи до файлів відносно сховища термів. За допомогою кеша у вигляді ETS таблиці ми убезпечуємо себе від подвійної нормалізації термів з одними і тими самими іменами.

```
(:star,N)       D → (:star,N+1)
(:var,N,I)      D → :true = proplists:is_defined N B, om:keyget N D I
(:remote,N)     D → om:cache (type N D)
(:pi,N,0,I,O)   D → (:star,h(star(type I D)),star(type O [(N,norm I)|D]))
(:fn,N,0,I,O)   D → let star (type I D), NI = norm I
                     in (:pi,N,0,NI,type(O,[(N,NI)|D]))
(:app,F,A)      D → let T = type(F,D),
                         (:pi,N,0,I,O) = T, :true = eq I (type A D)
                     in norm (subst O N A)
```

2.9 Зсув індексів де Брейна

Зсув виконує переіменування змінної N в термі B. Переіменування означає додавання одиниці до індексу де Брейна у компоненті яка індексується іменем змінної.

```
(:star,X)         N P -> (:star,X)
(:var,N,I)        N P -> (:var,N,I+1) when I >= P
                      -> (:var,N,I)
(:remote,X)       N P -> (:remote,X)
(:pi,N,0,I,O)     N P -> (:pi,N,0,sh I N P,sh O N P+1)
(:fn,N,0,I,O)     N P -> (:fn,N,0,sh I N P,sh O N P+1)
(:app,L,R)        N P -> (:app,L,R)
```

2.10 Підстановка

Операція підстановки рекрсивно підсталяє значення певної змінної, яка знаходиться в перному термі.

```
(:star,X)       N V L -> (:star,X)
(:var,N,L)      N V L -> V
(:var,N,I)      N V L -> (:var,N,I-1) when I > L
(:remote,X)     N V L -> (:remote,X)
(:pi,N,0,I,O)   N V L -> (:pi,N,0,sub I N V L,sub O N (sh V N 0) L+1)
(:pi,F,X,I,O)   N V L -> (:pi,F,X,sub I N V L,sub O N (sh V F 0) L)
(:fn,N,0,I,O)   N V L -> (:fn,N,0,sub I N V L,sub O N (sh V N 0) L+1)
(:fn,F,X,I,O)   N V L -> (:fn,F,X,sub I N V L,sub O N (sh V F 0) L)
(:app,F,A)      N V L -> (:app, sub F N V L,sub A N V L)
```

2.11 Нормалізація

Функція нормалізації рекурсивно виконує операцію підстановки для операції функціональної аплікаї (в літературі називається бета редукція, нормалізація за допомою обчислення та підстановки). Normalization performs substitutions on applications to functions (beta-reduction).

```
norm (:star,X)       -> (:star,X)
     (:var,X)        -> (:var,X)
     (:remote,N)     -> cache (norm N [])
     (:pi,N,0,I,O)   -> (:pi,N,0,norm I,norm O)
     (:fn,N,0,I,O)   -> (:fn,N,0,norm I,norm O)
     (:app,F,A)      -> case norm F of
                           (:fn,N,0,I,O) -> norm (subst O N A)
                                      NF -> (:app,NF,norm A) end
```

2.12 Рівність

Операція рівності за визначення просто рекурсивно порівнює два терма-дерева за допомогою патерн-мачінг оператора Erlang.

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

3 Використання мови

Продемонструємо інтерфейс користувача та покажемо на прикладах, як використовувати мову **PTS**$^{\infty}$. Перший приклад, це наївне намагання імплементувати MLTT примітиви за домогою PTS та лише П-типу. Ми будемо викорисовувати для нього кодування Бома-Берардуччі [3]. Другий приклад показує як писати реальні програми з вводом-виводом в середовищі виконання Ertlang. Ми покажемо формалізацію як індуктивних так і коіндуктивних процесів.

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

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

3.1 Сігма тип

The PTS system is extremely powerful even without **Sigma** type. But we can encode **Sigma** type similar how we encode **Prod** tuple pair in Bohm encoding. Let's formulate **Sigma** type as an inductive type in higher language.

```
data Sigma (A: Type) (P: A -> Type) (x: A): Type =
     (intro: P x -> Sigma A P)
```

The **Sigma-type** with its eliminators appears as example in Aaron Stump [15]. Here we will show desugaring to **PTS**$^{\infty}$.

```
-- Sigma/@
   \ (A: *)
-> \ (P: A -> *)
-> \ (n: A)
-> \/ (Exists: *)
-> \/ (Intro: A -> P n -> Exists)
-> Exists

-- Sigma/Intro
   \ (A: *)
-> \ (P: A -> *)
-> \ (x: A)
-> \ (y: P x)
-> \ (Exists: *)
-> \ (Intro: \/ (x:A) -> P x -> Exists)
-> Intro x y

-- Sigma/fst
   \ (A: *)
-> \ (B: A -> *)
-> \ (n: A)
-> \ (S: #Sigma/@ A B n)
-> S A ( \(x: A) -> \(y: B n) -> x)

-- Sigma/snd
   \ (A: *)
-> \ (B: A -> *)
-> \ (n: A)
-> \ (S: #Sigma/@ A B n)
-> S (B n) ( \(_: A) -> \(y: B n) -> y )

> om:fst(om:erase(om:norm(om:a("#Sigma/test.fst")))).
{{λ,{'Succ',0}},
 {any,{{λ,{'Zero',0}},{any,{var,{'Zero',0}}}}}}
```

For using **Sigma** type for Logic purposes one should change the home Universe of the type to **Prop**. Here it is:

```
data Sigma (A: Prop) (P: A -> Prop): Prop =
     (intro: (x:A) (y:P x) -> Sigma A P)
```

3.2 Тип рівності

Други приклад потужності мови з однією аксіомою це кодування типа рівності системи типів Мартіна-Льофа.

```
data Equ (A: Type): A -> A -> Type :=
          (refl (a: A): Equ A a a)

-- Equ/@
    \ (A: *)
-> \ (x: A)
-> \ (y: A)
-> \/ (Equ: A -> A -> *)
-> \/ (Refl: \/ (z: A) -> Equ z z)
-> Equ x y

-- Equ/Refl
    \ (A: *)
-> \ (x: A)
-> \ (Equ: A -> A -> *)
-> \ (Refl: \/ (z: A) -> Equ z z)
-> Refl x
```

You cannot construct a lambda that will check different values of A type is they are equal, however, you may want to use built-in definitional equality and normalization feature of type checker to actually compare two values:

```
> om:print(om:type(
  om:a("(\\ (z: #Equ/@ #Nat/@ #Nat/One #Nat/One) -> #Prop/True)"++
        " (#Equ/Refl #Nat/@ (#Nat/Succ #Nat/Zero))"))).
    \/ (True: *0)
-> \/ (Intro: True)
-> True
ok

> om:print(om:type(
  om:a("(\\ (z: #Equ/@ #Nat/@ #Nat/One #Nat/One) -> #Prop/True)"++
        " (#Equ/Refl #Nat/@ #Nat/Zero)"))).
** exception error: no match of right hand side value
    {error,{"==",
            {app,{{var,{'Succ',0}},{var,{'Zero',0}}}},
            {var,{'Zero',0}}}}
```

3.3 Система ефектів

This work is expected to compile to a limited number of target platforms. For now, Erlang, Haskell, and LLVM are awaiting. Erlang version is expected to be used both on LING and BEAM Erlang virtual machines. This language allows you to define trusted operations in System F and extract this routine to Erlang/OTP platform and plug as trusted resources. As the example, we also provide infinite coinductive process creation and inductive shell that linked to Erlang/OTP IO functions directly.

IO protocol. We can construct in pure type system the state machine based on (co)free monads driven by **IO/IOI** protocols. Assume that **String** is a **List Nat** (as it is in Erlang natively), and three external constructors: getLine, putLine and pure. We need to put correspondent implementations on host platform as parameters to perform the actual IO.

```
String: Type = List Nat
data IO: Type =
      (getLine: (String -> IO) -> IO)
      (putLine: String -> IO)
      (pure: () -> IO)
```

3.3.1 Нескінченний ввід-вивід

Типова специфікація на тип Infinity I/O для нескінченного вводу-виводу.

```
-- IOI/@: (r: U) [x: U] [[s: U] -> s -> [s -> #IOI/F r s] -> x] x
   \ (r : *)
-> \/ (x : *)
-> (\/ (s : *)
   -> s
   -> (s -> #IOI/F r s)
   -> x)
-> x

-- IOI/F
   \ (a : *)
-> \ (State : *)
-> \/ (IOF : *)
-> \/ (PutLine_ : #IOI/data -> State -> IOF)
-> \/ (GetLine_ : (#IOI/data -> State) -> IOF)
-> \/ (Pure_ : a -> IOF)
-> IOF

-- IOI/MkIO
   \ (r : *)
-> \ (s : *)
-> \ (seed : s)
-> \ (step : s -> #IOI/F r s)
-> \ (x : *)
-> \ (k : forall (s : *) -> s -> (s -> #IOI/F r s) -> x)
-> k s seed step

-- IOI/data
#List/@ #Nat/@
```

Приклад нескінченного вводу-виводу.

```
-- Morte/corecursive
( \ (r: *1)
 -> ( (((#IOI/MkIO r) (#Maybe/@ #IOI/data)) (#Maybe/Nothing #IOI/data))
    ( \ (m: (#Maybe/@ #IOI/data))
     -> (((((#Maybe/maybe #IOI/data) m) ((#IOI/F r) (#Maybe/@ #IOI/data)))
           ( \ (str: #IOI/data)
            -> ((((#IOI/putLine r) (#Maybe/@ #IOI/data)) str)
                 (#Maybe/Nothing #IOI/data))))
         (((#IOI/getLine r) (#Maybe/@ #IOI/data))
          (#Maybe/Just #IOI/data))))))
```

Коіндуктивні біндінги в Erlang.

```
copure() ->
    fun (_) -> fun (IO) -> IO end end.

cogetLine() ->
    fun(IO) -> fun(_) ->
        L = ch:list(io:get_line("> ")),
        ch:ap(IO,[L]) end end.

coputLine() ->
    fun (S) -> fun(IO) ->
        X = ch:unlist(S),
        io:put_chars(": "++X),
        case X of "0\n" -> list([]);
                        _ -> corec() end end end.

corec() ->
    ap('Morte':corecursive(),
        [copure(),cogetLine(),coputLine(),copure(),list([])]).


> om_extract:extract("priv/normal/IOI").
ok
> Active: module loaded: {reloaded,'IOI'}

> om:corec().
> 1
: 1
> 0
: 0
#Fun<List.3.113171260>
```

3.3.2 Скінченний ввід-вивід

Типова специфікація індуктивного типу I/O для скінченного вводу-виводу.

```
-- IO/@
    \ (a : *)
-> \/ (IO : *)
```

```
-> \/ (GetLine_ : (#IO/data -> IO) -> IO)
-> \/ (PutLine_ : #IO/data -> IO -> IO)
-> \/ (Pure_ : a -> IO)
-> IO

-- IO/replicateM
   \ (n: #Nat/@)
-> \ (io: #IO/@ #Unit/@)
-> #Nat/fold n (#IO/@ #Unit/@)
               (#IO/[>>] io)
               (#IO/pure #Unit/@ #Unit/Make)
```

Приклад скінченної рекурсивної програми.

```
-- Morte/recursive
((#IO/replicateM #Nat/Five)
 ((((#IO/[>>=] #IO/data) #Unit/@) #IO/getLine) #IO/putLine))
```

Індуктивні біндінги в Erlang.

```
pure() ->
    fun(IO) -> IO end.

getLine() ->
    fun(IO) -> fun(_) ->
        L = ch:list(io:get_line("> ")),
        ch:ap(IO,[L]) end end.

putLine() ->
    fun (S) -> fun(IO) ->
        io:put_chars(": "++ch:unlist(S)),
        ch:ap(IO,[S]) end end.

rec() ->
    ap('Morte':recursive(),
        [getLine(),putLine(),pure(),list([])]).
```

Приклад виконання рекурсивної програми всередині обчислювального середовища Erlang.

```
> om:rec().
> 1
: 1
> 2
: 2
> 3
: 3
> 4
: 4
> 5
: 5
#Fun<List.28.113171260>
```

4 Вища мова з індуктивними типами

As was shown by Herman Geuvers [9] the induction principle is not derivable in second-order dependent type theory. However there a lot of ways doing this. For example, we can build in induction principal into the core for every defined inductive type. We even can allow recursive type check for only terms of induction principle, which have recursion base — that approach was successfully established by Peng Fu and Aaron Stump [15]. In any case for derivable induction principle in PTS$^{\infty}$ we need to have fixpoint somehow in the core.

So-called Calculus of Inductive Constructions [14] is used as a top language on top of PTS to reason about inductive types. Here we will show you a sketch of such inductive language model which intended to be a language extension to PTS system. CiC is allowing fixpoint for any terms, and base checking should be performed during type checking such terms.

Our future top language is a general-purpose functional language with Π and Σ types, recursive algebraic types, higher order functions, corecursion, and a free monad to encode effects. It compiles to a small MLTT core of dependent type system with inductive types and equality. It also has an Id-type (with its recursor) for equality reasoning, Case analysis over inductive types.

4.1 БНФ

```
<> ::= #option
[] ::= #list
 | ::= #sum
 1 ::= #unit
 I ::= #identifier
 U ::= Type < #nat >
 T ::= 1 | ( I : O ) T
 F ::= 1 | I : O = O , F
 B ::= 1 | [ | I [ I ] → O ]
 O ::= I | ( O ) |
       U | O → O                | O O
         | fun ( I : O ) → O    | fst O
         | snd O                | id O O O
         | J O O O O O          | let F in O
         | ( I : O ) * O        | ( I : O ) → O
         | data I T : O := T    | record I T : O := T
         | case O B
```

4.2 Синтаксичне дерево

The AST of higher language is formally defined using itself. Here you can find telescopes (context lists), split and its branches, inductive data definitions.

```
data tele (A: U)   = emp | tel (n: name) (b: A) (t: tele A)
data branch (A: U) =        br (n: name) (args: list name) (term: A)
data label (A: U)  =       lab (n: name) (t: tele A)
data ind
   = star                            (n: nat)
   | var    (n: name)                (i: nat)
   | app              (f a: ind)
   | lambda (x: name) (d c: ind)
   | pi     (x: name) (d c: ind)
   | sigma  (n: name) (a b: ind)
   | arrow            (d c: ind)
   | pair             (a b: ind)
   | fst              (p:   ind)
   | snd              (p:   ind)
   | id               (a b: ind)
   | idpair           (a b: ind)
   | idelim           (a b c d e: ind)
   | data_  (n: name) (t: tele ind) (labels:   list (label ind))
   | case   (n: name) (t: ind)      (branches: list (branch ind))
   | ctor   (n: name)               (args:     list ind)
```

The Erlang version of parser encoded with OTP library **yecc** which implements LALR-1 grammar generator. This version resembles the model and slightly based on cubical type checker by Mortberg [4] and could be reached at Github repository[4].

4.3 Кодування індуктивних типів

There are a number of inductive type encodings: 1) Commutative square encoding of F-algebras by Hinze, Wu [11]; 2) Inductive-recursive encoding, algebraic type of algebraic types, inductive family encoding by Dagand [7]; 3) Encoding with motives inductive-inductive definition, also with inductive families, for modeling quotient types by Altenkirch, Kaposi [1]; 4) Henry Ford encoding or encoding with Ran,Lan-extensions by Hamana, Fiore [10]; 5) Church-compatible Bohm-Berarducci encoding Bohm, Berarducci [3]. Om is shipped with base library in Church encoding and we already gave the example of IO system encoded with runtime linkage. We give here simple calculations behind this theory.

[4] http://github.com/groupoid/infinity/tree/master/priv

4.4 Поліноміальні функтори

Least fixed point trees are called well-founded trees. They 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: $\mu\ X \rightarrow \mu\ Y \rightarrow 1 + X \times Y = \mu\ X = List\ X$

As we know there are several ways to appear for a variable in a recursive algebraic type. Least fixpoint is known as a recursive expression that has a base of recursion In Chuch-Bohm-Berarducci encoding type are store as non-recursive definitions of their right folds. A fold in this encoding is equal to id function as the type signature contains its type constructor as parameters to a pure function.

4.5 Приклад кодування модуля List

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

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

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

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

Here traditionally we show the **List** definition in higher language and its desugared version in **Om** language.

```
data List: (A: *) → * :=
     (Cons: A → list A → list A)
     (Nil: list A)
```

```
-- List/@
   \ (A : *)
-> \/ (List: *)
-> \/ (Cons: \/ (Head: A) -> \/ (Tail: List) -> List)
-> \/ (Nil: List)
-> List

-- List/Cons
   \ (A: *)
-> \ (Head: A)
-> \ (Tail:
        \/ (List: *)
     -> \/ (Cons: \/ (Head: A) -> \/ (Tail: List) -> List)
     -> \/ (Nil: List)
     -> List)
-> \ (List: *)
-> \ (Cons:
        \/ (Head: A)
     -> \/ (Tail: List)
     -> List)
-> \ (Nil: List)
-> Cons Head (Tail List Cons Nil)

-- List/Nil
   \ (A: *)
-> \ (List: *)
-> \ (Cons:
        \/ (Head: A)
     -> \/ (Tail: List)
     -> List)
-> \ (Nil: List)
-> Nil
```

```
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))
```

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

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

4.6 Базова бібліотека

Базова бібліотека включає основні теоретико-типові конструкції, такі як **Unit**, **Bool**, **Either**, **Maybe**, **Nat**, **List** та **IO**. Покажемо на прикладах, як це виглядає. Повна версія базової бібліотеки доступна на Github за адресою [5].

```
  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 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))
```

4.7 Вимірювання та порівняння з іншими системами

Мова типового верифікатора **PTS**$^\infty$ у свою чергу є цільовою мовою для мов більш високого рівня, як то мова з індуктивними типами або мова з гомотопічним відрізком. Загальний розмір бібліотеки мови **PTS**$^\infty$ складає 300 рядків.

[5] http://github.com/groupoid/pts

Табл. 2: Compiler Passes

Module	LOC	Description
om_tok	54 LOC	Токенайзер
om_parse	81 LOC	Синтаксичний парсер
om_type	60 LOC	Нормалізатор та верифікатор
om_erase	36 LOC	Стирання типів
om_extract	34 LOC	Екстракція Erlang байткоду

5 Висновки

В роботі запропонована модифікована версія CoC, також відома як чиста система або система з однією аксіомою, разом з предикативною або імпредикативною ієрархією зліченної кількості всесвітів. Ця система відома як консистентна, підтримує сильну нормалізацію термів, та відвторює базове ядро усіх сучасних систем доведення теорем, таких як Coq, Lean, Agda, тощо.

Результати дослідження В результаті цього дослідження встановлені наступні відкриття: 1) Відсутність рекурсії робить неможливим кодування принципу індукції, необхідне послаблення хоча б для сигнатур рекурсивних типів, так звані Self-типи, запропоновані у мові Cedile, авторами Peng Fu та Aaron Stump [8]. 2) Однак для запезпечення виконання програм System F цілком достатньо для забезпечення бібліотеки часу виконання; 2) Теореми які можуть бути вираженими без конструктора нерухомої точки, більше відповідають категорній семантиці; 3) Ця система може бути природнім чином транпортована у нетипизовані лямбда числення та їх евалуатори, що розширює область застосування майже на усі інтерпретатори. 4) Якщо обмежити розмір інтерпретатор та його програм розміром кеша першого рівня, то швидкість інтерпретації лямбда контекстів буде на рівні JIT або native.

Переваги над існуючими аналогічними рішеннями. 1) рафінована версія типового верифікатора на 300 рядків; Мінімальність ядра дає змогу швидко переконатися в коректності та здійснити ревью; 2) підтримка предикативної та імпредикативної ієрархій, як елемент конфігурації $\mathbf{PTS}^{\infty}$; 3) евалуатор Erlang більш ефективний ніж вбудовування в Haskell; 4) мова $\mathbf{PTS}^{\infty}$ використувється для специфікації нескінченних процесів.

Наукове та виробниче використання. 1) Ця мова може використовуватися як сертифікована ядро для додатків з підвищеними вимогами до якості, такі як фінансові додатки, математичні, або інші теми, які потребують перевірки тотальності функцій; 2) Ця мова може використовуватися як вбудовувана бібліотека; 3) В академії $\mathbf{PTS}^{\infty}$ може використовуватися як дидактичний інструмент з логіки, систем типів, лямбда численню, функціональним мовам програмування.

Перспективи подальших досліджень. 1) Розширення спектру цільових мов та доведення семантики у cubicaltt; 2) Побудова екстрактор в опмитальний лямбда евалуатор з $\mathbf{PTS}^{\infty}$; 3) Побудова сертифікованого інтерпретатора на Rust як заміна Erlang; 4) Залучення принципу індукції в $\mathbf{PTS}^{\infty}$ в майбутньому.

6 Подяки

Висловлюється подяка усім дописувичам Групоїд Інфініті, хто допоміг уникнути помилок в TeX та Erlang файлах. Також подяка всім рідним, хто підтримує нас.

Література

[1] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 18–29, New York, NY, USA, 2016. ACM.

[2] H. P. Barendregt. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309, New York, NY, USA, 1992. Oxford University Press, Inc.

[3] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. In *Theoretical Computer Science*, volume 39, pages 135–154, 1985.

[4] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. In *Cubical Type Theory: a constructive interpretation of the univalence axiom*, volume abs/1611.02108, 2017.

[5] Thierry Coquand. An algorithm for type-checking dependent types. In *Sci. Comput. Program.*, volume 26, pages 167–177, 1996.

[6] Thierry Coquand and Gerard Huet. The calculus of constructions. In *Information and Computation*, pages 95–120, Duluth, MN, USA, 1988. Academic Press, Inc.

[7] P.É. Dagand, University of Strathclyde. Department of Computer, and PhD thesis Information Sciences. *A Cosmology of Datatypes: Reusability and Dependent Types.* 2013.

[8] Peng Fu and Aaron Stump. Self types for dependently typed lambda encodings. In *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 224–239, 2014.

[9] Herman Geuvers. *Induction Is Not Derivable in Second Order Dependent Type Theory*, pages 166–181. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[10] Makoto Hamana and Marcelo P. Fiore. A foundation for gadts and inductive families: dependent polynomial functor approach. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 59–70, 2011.

[11] Ralf Hinze and Nicolas Wu. Histo- and dynamorphisms revisited. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, WGP '13, pages 1–12, New York, NY, USA, 2013. ACM.

[12] Simon Peyton Jones and Erik Meijer. Henk: A typed intermediate language. In *In Proc. First Int'l Workshop on Types in Compilation*, 1997.

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

[14] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

[15] Aaron Stump. The calculus of dependent lambda eliminations. In *Journal of Functional Programming*, volume 27. Cambridge University Press, 2017.