Volume II: Systems

Introduction to System Programming

Namdak Tonpa

2022 · Groupoid Infinity

II

Зміст

Issue VI: The Arthur Language

Maksym Sokhatskyi [1]

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute
1 травня 2025 р.

Анотація

Minimal language for parallel computations in symmetric monoidal categories.
Keywords: Interaction Networks, Symmetric Monoidal Categories

Присвячується автору K

Артуру Вітні

Третій розділ описує розвиток концептуальної моделі системи доведення теорем як сукупності формальних середовищ виконання, кожне наступе з яких, складніше за попереднє, має свою операційну семантику, та наслідує усі властивості попередніх операційних середовищ послідовності.

1 The Arthur Language

Мінімальна мова системи O_{CPS_λ} визначається простим синтаксичним деревом:

```
def CPS$_\lambda$ : U
 := inductive { var (x: nat)
              | lam (l: nat) (d: cps)
              | app (f a: cps)
              }
```

Однак, на практиці, застосовують більш складні описи синтаксичних дерев, зокрема для лінивих обчислень, та розширення синтаксичного дерева спеціальними командами пов'язаними з середовищем виконання. Програми таких інтерпретаторів відповідно виконуються у певній пам'яті, яка використовується як контекст виконання. Кожна така програма крутиться як одиниця виконання на певному ядрі процесора. Ситема процесів, де кожен процес є CPS-програмою яку виконує інтерпретатор на певному ядрі.

Мотивація для побудови такого інтерпретатору, який повністю розміщується разом зі программою в L1 стеку (який лімітований 64КБ) базується на успіху таких віртуальних машин як LuaJIT, V8, HotSpot, а також векторних мов програмування типу K та J. Якби ми могли побудувати дійсно швидкий інтерпретатор який би виконував програми цілком в L1 кеші, байткод та стріми якого були би вирівняні по словам архітектури, а для векторних обчислень застосовувалися би AVX інструкції, які, як відомо перемагають по ціні-якості GPU обчислення. Таким чином, такий інтерпретатор міг би, навіть без спеціалізованої JIT компіляції, скласти конкуренцію сучасним промисловим інтерпретаторам, таким як Erlang, Python, K, LuaJIT.

Для дослідження цієї гіпотези мною було побудовано еспериментальний інтерпретатор без байт-коду, але з вирівняним по словам архітерктури стріму команд, які є безпосередньою машинною презентацією конструкторів індуктивних типів (enum) мови Rust. Наступні результати були отримані після неотпимізованої версії інтерпретатора при обчисленні факторіала (5) та функції Акермана у точці (3,4).

Ключовим викликом тут стали лінійні типи мови Rust, які не дозволяють звертатися до ссилок, які вже були оброблені, а це впливає на всю архітектуру тензорного преставлення змінних в мові інтерпретатор O_{CPS}, яка наслідує певним чином мову K.

Табл. 1: Заміри на інтерпретаторах ландшафту атаки

Мова	Fac(5) в нс
Rust	0
Java	3
PyPy	8
CPS	291
Python	537
K	756/635
Erlang	10699/1806/436/9
LuaJIT	33856

Табл. 2: Заміри на інтерпретаторах ландшафту атаки

Мова	Akk(3,4) в мкс
CPS	635
Rust	8,968

1.1 Векторизація засобами мови Rust

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

1.2 Байт-код інтерпретатора

Синтаксичне дерево, або неформалізований бай-код віртуальної машини або інтерпретатора O_{CPS} розкладається на два дерева, одне дерево для управляючих команд інтерпретатора: Defer, Continuation, Start (початок програми), Return (завершення програми).

```
def Lazy : U
 := inductive { Defer (otree: NodeId) (a: AST) (cont: Cont)
              | Continuation (otree: NodeId) (a: AST) (cont: Cont)
              | Return (a: AST)
              | Start
              }
```

Операції віртуальної машини: умовний оператор, оператор присвоєння, лямбда функція та аплікація, є відображеннями на конструктори синтаксичного дерева.

```
def Cont : U
 := inductive { Expressions (a: AST) (v: Option (Iter AST)) (c: Cont)
              | Assign (ast: AST) (cont: Cont)
              | Cond (c,d: AST) (cont: Cont)
```

```
| Func (a,b,c: AST) (cont: Cont)
| List (acc: Vec AST) (vec: Iter AST) (i: Nat) (c: Cont)
| Call (a: AST) (i: Nat) (cont: Cont)
| Return
| Intercore (m: Message) (cont: Cont)
| Yield (cont: Cont)
}
```

1.3 Синтаксис

Синтаксис мови O_{CPS} підтримує тензори, та звичайне лямбда числення з значеннями у тензорах машинних типів даних: i32, i64.

```
E: V | A | C
NC: ";" = [] | ";" m:NL = m
FC: ";" = [] | ";" m:FL = m
EC: ";" = [] | ";" m:EL = m
NL: NAME | o:NAME m:NC = Cons o m
FL: E | o:E | m:FC = Cons o m
EL: E | EC  | o:E m:EC = Cons o m
C: N | c:N a:C = Call c a
N: NAME | S | HEX | L | F
L: "(" ")" = [] | "([" c:NL "]" m:FL ")" = Table c m
                | "(" l:EL ")" = List l
F: "{" "}" = Lambda [] [] []
            | "{[" c:NL "]" m:EL "}" = Lambda [] c m
            | "{" m:EL "}" = Lambda [] [] m
```

Після парсера, синтаксичне дерево розкладається по наступним складовим: AST для тензорів (визначення вищого рівня); Value для машинних слів; Scalar для конструкцій мови (куди входить зокрема списки та словники, умовний оператор, присвоєння, визначення функції та її аплікація, UTF-8 літерал, та оператор передачі управління в поток планувальника який закріплений за певним ядром CPU).

```
def AST : U
 := inductive { Atom (a: Scalar)
              | Vector (a: Vec AST)
              }

def Value : U
 := inductive { Nil
              | SymbolInt (a: u16)
              | SequenceInt (a: u16)
              | Number (a: i64)
              | Float (a: f64)
              | VecNumber (Vec i64)
              | VecFloat (Vec f64)
              }
```

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

Кожна секція цієї глави буде присвячена цим мовним компонентам системи доведення теорем. В кінці розділу дається повна система, яка включає в себе усі мови та усі мовні перетворення.

1.4 Система числення процесів SMP async

1.4.1 Операційна система

Перелічимо основні властивості операційної системи (прототип якої опублікований на Github[1]).

Властивості: автобалансована низьколатентна, неблокована, без копіювання, система черг з CAS-мультикурсорами, з пріоритетами задач та масштабованими таймерами.

1.4.2 Асиметрична багапроцесорність

Ядро системи використовує асиметричну багапроцесорність (АП) для планування машинного часу. Так у системі для консольного вводу-виводу та вебсокет моніторингу використовується окремий ректор (закріплений за ядром процесора), аби планування не впливало на програми на інших процесорах.

[1] `https://github.com/voxoz/kernel`

Це означає статичне закріплення певного атомарного процесу обчислення за певним реактором, та навіть можливо дати гарантію, що цей процес не перерветься при наступному кванті планування ніяким іншим процесом на цьому ядрі (ситуація єдиного процесу на реактор ядра процесору). Ядро системи постачається разом з конфігураційною мовою для закріплення задач за реакторами:

```
reactor[aux;0;mod[console;network]];
reactor[timercore;1;mod[timer]];
reactor[core1;2;mod[task]];
reactor[core2;3;mod[task]];
```

1.4.3 Низьколатентність

Усі реактори повинні намагатися обмежити IP-лічильник команд діапазоном розміром з L1/L2 кеш об'єм процесора, для унеможливлення колізій між ядрами на міжядерній шині можлива конфігурація, де реактори виконують код, області пам'яті якого не перетинаються, та обмежені об'ємом L1 кеш пам'яті що при наявній AVX векторизації дать змогу повністю використовувати ресурси процесору наповну.

1.4.4 Мультикурсори

Серцем низьколатентної системи транспорту є система наперед виділений кільцевих буферів (які називаються секторами глобального кільця). У цій системі кілець діє система курсорі для запису та читання, ці курсори можуть мати різний напрямок руху. Для забезпечення імутабельності (нерухомості

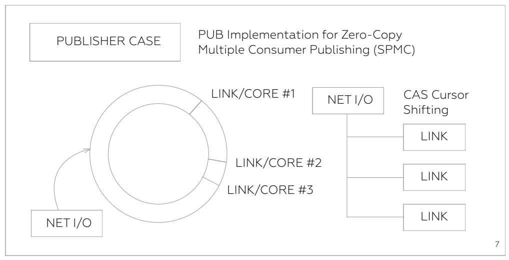

Рис. 1: Кільцева статична черга з CAS-курсором для публікації

даних) та відсутності копіювання в подальшій роботі, дані залишаються в черзі, а рухаються та передаються лише курсори на типизовані послідовності даних.

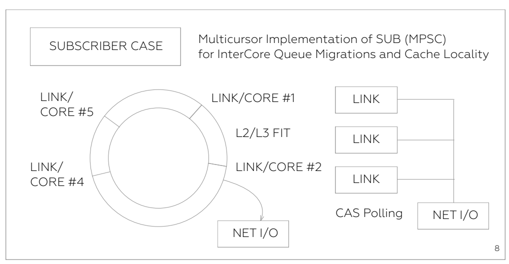

Рис. 2: Кільцева статична черга з CAS-курсором для згортки

1.4.5 Реактори

Кожен процесор має три типи реакторів які можуть бути на ньому запущені: i) Task-реактор; ii) Timer-реактор; iii) IO-цикли. Для Task-реактора існують черги пріорітетів, а для Timer-реактора — дерева інтервалів. Загальний

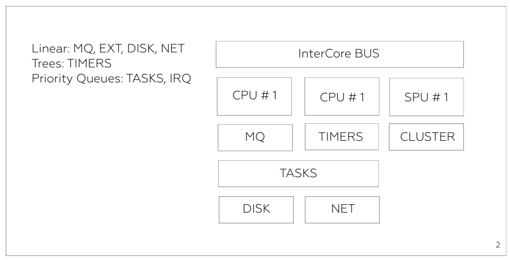

Рис. 3: Система процесорних ядер та реакторів

спосіб комунікації для задач виглядає як публікація у чергу (рух курсора запису) та підписка на черги і згортання (руху курсора читання). Кожна черга має як курсори для публікації так і курсори для читання. Можливо також використання міжреакторної шини InterCore та посилання службового повідомлення по цій шині на інший реактор. Так, наприклад, працюють таймери та старти процесів, які передають сигнал в реактор для перепланування. Можна створювати нові повідомлення шини InterCore і систему фільтрів для згортання черги реактора для більш гнучкої обробки сигналів реального часу.

Task-реактор

Task-реактор або реактор задач виконує Rust задачі або програми інтерпретатора, які можуть бути двох видів: кінечні (які повертають результат виконання), або нескінченні (процеси).

Приклад бескінечної задачі — 0-процес, який запускається при старті системи. Цей процес завжди доступний по WebSocket каналу та з консолі терміналу.

IO-реактор

Мережевий сервер або IO-реактор може обслуговувати багато мережевих з'єднань та підтримує Windows, Linux, Mac смаки.

Timer-реактор

Різні типи сутностей планування (такі як Task, IO, Timer) мають різні дисципліни селекторів повідомлень для черг (послідовно, через само-балансуючі дерева, BTree дерева тощо).

1.4.6 Міжреакторний транспорт InterCore

Шина InterCore конструюється певним числом SPMC черг, виділених для певного ядра. Шина сама має топологію зірки між ядрами, та черга MPSC організована як функція над множиною паблішерів. Кожне ядро має рівно одного паблішера. Функція обробки шини протоколу InterCore називається poll_bus та є членом планувальника. Ви можете думати про InterCore як телепорт між процесорами, так як pull_bus викликається після кожної операції Yield в планувальник, і, таким чином, якщо певному ядру опублікували в його чергу повідомлення, то після наступного Yield на цьому ядрі буде виконана функція обробки цього повідомлення.

```
fun pub(capacity: int ): int
```

Створює новий CAS курсор для паблішінга, тобто для запису. Повертає глобальних машинний ідентифікатор, має єдиний параметр, розмір черги. Приклад: p: pub[16].

```
fun sub(publisher: int ): int
```

Створює новий CAS курсор для читання певної черги, певного врайтера. Повертає глобальний машинний ідентифікатор для читання. Приклад: s: sub[p].

```
fun spawn(core: int , program: code, cursors: array int ): int
```

Створює нову програму задачу CPS-інтепреторатора для певного ядра. Задача може бути або програмою на мові Rust або будь якою програмою через FFI. Також при створенні задачі задається список курсорів, які ексклюзивно належатимуть до цієї задачі. Параметри функції: ядро, текст програми або назва FFI функції,спсисок курсорів.

```
fun kill (process: int ): int
```

Денонсація процесора на реаторі.

```
fun send(writer: int , data: binary): int
```

Посилає певні дані в певний курсор для запису. Повертає Nil якшо всьо OK. Приклад: snd[p;42].

```
fun receive(reader: int)
```

Повертає прочитані дані з певного курсору. Якшо даних немає, то передає управління в планувальних за допомогою Yield. Приклад: rcv[s].

1.5 Структури ядра

Ядро є ситемою акторів з двома основними типами акторів: чергами, які представляють кільцеві буфери та відрізки памяті; та задачами, які резпрезентують байт-код програм та їх інтерпретацію на процесорі. Черги бувають двох видів: для публікації, які місять курсори для запису; та для читання, які містять курсори для читання. Задачі можна імплементувати як Rust програми, або як O_{CPS} програми.

1.5.1 Черга для публікації

```
pub struct Publisher<T> {
    ring: Arc<RingBuffer<T>>,
    next: Cell<Sequence>,
    cursors: UncheckedUnsafeArc<Vec<Cursor>>,
}
```

1.5.2 Черга для читання

```
pub struct Subscriber<T> {
    ring: Arc<RingBuffer<T>>,
    token: usize,
    next: Cell<Sequence>,
    cursors: UncheckedUnsafeArc<Vec<Cursor>>,
}
```

Існує дві спеціальні задачі: InterCore задача, написана на Rust, яка запускається на всіх ядрах при запуску системи, а також CPS-інтерпретор головного термінала системи, який запускається на BSP ядрі, поблизжче до Console та WebSocket IO селекторів. В процесі життя різні CPS та Rust задачі можуть бути запущені в такій системі, поєднуючи гнучкість програм інтерпретатора, та низькорівневих програм, написаних на мові Rust.

Окрім черг та задач, в системі присутні також таймери та інші IO задачі, такі як сервери мережі або сервери доступу до файлів. Також існують структури які репрезентують ядра та містять палнувальники. Уся віртуальна машина є сукупністю таких структур-ядер.

1.5.3 Канал

Канал складається з одного курсору для запису та багатьох курсорів для читання. Канал предствляє собою компонент зірки шини InterCore.

```
pub struct Channel {
    publisher: Publisher<Message>,
    subscribers: Vec<Subscriber<Message>>,
}
```

1.5.4 Черги ядра

Память репрезентує усі наявні черги для публікації та читання на ядрі. Ця інформація передається клонованою кожній задачі планувальника на цьому ядрі.

```
pub struct Memory<'a> {
    publishers: Vec<Publisher<Value<'a>>>,
    subscribers: Vec<Subscriber<Value<'a>>>,
}
```

1.5.5 Планувальник

Планувальник репрезентує ядра процесара, які розрізняються як BSP-ядра (або 0-ядра, bootstrap) та AP ядра (інші ядра > 0, application). BSP ядро тримає на собі Console та WebSocket IO селектори. Це означає, що BSP ядро дає свій час на обробку зовнішньої інформації, у той час як AP процесори не обтяжені таким навантаженням (io черга в таких планувальниках пуста). Існує InterCore повідомлення яке додає або видаляє довільні IO селектори в планувальних для довільних конфігурацій.

```
pub struct Scheduler<'a> {
    pub tasks: Vec<T3<Job<'a>>>,
    pub bus: Channel,
    pub queues: Memory<'a>,
    pub io: IO,
}
```

1.5.6 Протокол InterCore

Протокол шини InterCore.

```
pub enum Message {
    Pub(Pub),
    Sub(Sub),
    Print(String),
    Spawn(Spawn),
    AckSub(AckSub),
    AckPub(AckPub),
    AckSpawn(AckSpawn),
    Exec(usize, String),
    Select(String, u16),
    QoS(u8, u8, u8),
    Halt,
    Nop,
}
```

1.6 Система числення тензорів AVX

Для реалізації мови програмування високого рівня на BLAS Level 3 бекендом була вибрана мова NumLin, серед інших: 1) Ling, 2) Guarded Cubical, 3) A Fibrational Framework for Substructural and Modal Logics, 4) APL-like interpreter in Rust (дана робота), 5) Futhark.

```
def AVX-512 : U
 := inductive { Star | True | False
              | Variable (_: Var)
              | Prim (_: Builtin)
              | Int (_: nat) | Float (_: float)
              | Lambda (a: Var) (b: Linear) (c: Exp)
              | App (a b: Exp)
              | Pair (a b: Var) (c d: Exp)
              | Consume (a: Var) (b c: Exp)
              | Gen (a: Var) (b: Exp)
              | Spec (a: Exp) (b: Fraction)
              | Fix (a b: Var) (c d: Linear) (e: Exp)
              | If (a b c: Exp)
              | Let (a: Var) (b c: Exp)
              }
```

1.7 Висновки

Перша стадія реалізації класичного лінивого інтерпретатора з CPS семантикою була виконана як MVP трейдингової HFT платформи. Наступна стадія — виконання верифікованого інтерпретатора (віртуальної машини) та компілятора (в неї) Standard ML мови на основі компілятора Joe (MinCaml).

Issue VII: Cartesian Interpreter

Maksym Sokhatskyi [1]

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute
1 травня 2025 р.

Анотація

Minimal language for sequential computations in cartesian closed categories.

Keywords: Lambda Calculus, Cartesian Closed Categories

Присвячується автору Erlang

Джо Армстронгу

2 The Joe Language

Мова програмування **Joe** — це чиста нетипізована мова, що є внутрішньою мовою декартово-замкнених категорій. Вона базується на лямбда-численні, розширеному парами, проєкціями та термінальним об'єктом, забезпечуючи мінімальну модель для обчислень у категорійному контексті.

2.1 Синтаксис

Терми **Joe** складаються зі змінних, лямбда-абстракцій, застосувань, пар, проєкцій (першої та другої) та термінального об'єкта. Це мінімальна мова, що підтримує обчислення через бета-редукцію та проєкції.

```
I = #identifier
O = I | ( O ) | O O | λ I -> O | O , O | O.1 | O.2 | 1

type term =
  | Var of string
  | Lam of string * term
  | App of term * term
  | Pair of term * term
  | Fst of term
  | Snd of term
  | Unit
```

2.2 Правила обчислень

Основними правилами обчислень у **Joe** є бета-редукція для лямбда-абстракцій та правила проєкцій для пар. Термінальний об'єкт є незвідним.

```
App (Lam (x, b), a) → subst x a b
Fst (Pair (t1, t2)) → t1
Snd (Pair (t1, t2)) → t2
```

$$\frac{(\lambda x.b)\ a}{b[a/x]} \quad \frac{\text{fst}\ \langle t_1, t_2\rangle}{t_1} \quad \frac{\text{snd}\ \langle t_1, t_2\rangle}{t_2}$$

2.3 Підстановка

```
let rec subst x s = function
  | Var y -> if x = y then s else Var y
  | Lam (y, t) when x <> y -> Lam (y, subst x s t)
```

```
  | App (f, a) -> App (subst x s f, subst x s a)
  | Pair (t1, t2) -> Pair (subst x s t1, subst x s t2)
  | Fst t -> Fst (subst x s t)
  | Snd t -> Snd (subst x s t)
  | Unit -> Unit
  | t -> t
```

2.4 Рівність

```
let rec equal t1 t2 =
  match t1, t2 with
  | Var x, Var y -> x = y
  | Lam (x, b), Lam (y, b') -> equal b (subst y (Var x) b')
  | Lam (x, b), t -> equal b (App (t, Var x))
  | t, Lam (x, b) -> equal (App (t, Var x)) b
  | App (f1, a1), App (f2, a2) -> equal f1 f2 && equal a1 a2
  | Pair (t1, t2), Pair (t1', t2') -> equal t1 t1' && equal t2 t2'
  | Fst t, Fst t' -> equal t t'
  | Snd t, Snd t' -> equal t t'
  | Unit, Unit -> true
  | _ -> false
```

2.5 Редукція

```
let rec reduce = function
  | App (Lam (x, b), a) -> subst x a b
  | App (f, a) -> App (reduce f, reduce a)
  | Pair (t1, t2) -> Pair (reduce t1, reduce t2)
  | Fst (Pair (t1, t2)) -> t1
  | Fst t -> Fst (reduce t)
  | Snd (Pair (t1, t2)) -> t2
  | Snd t -> Snd (reduce t)
  | Unit -> Unit
  | t -> t
```

2.6 Нормалізація

```
let rec normalize t =
  let t' = reduce t in
  if equal t t' then t else normalize t'
```

2.7 Внутрішня мова ДЗК

Мова **Joe** є внутрішньою мовою декартово-замкненої категорії (ДЗК). Вона включає лямбда-абстракції та застосування для замкнутої структури, пари та проєкції для декартового добутку, а також термінальний об'єкт для відновлення повної структури ДЗК.

Література

[1] Alonzo Church. *A Set of Postulates for the Foundation of Logic*. 1933.

[2] Alonzo Church. *An Unsolvable Problem of Elementary Number Theory*. 1941.

[3] Haskell Curry and Robert Feys. *Combinatory Logic, Volume I*. 1951.

[4] Dana Scott. *A Type-Free Theory of Lambda Calculus*. 1970.

[5] John C. Reynolds. *Towards a Theory of Type Structure*. 1974.

[6] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. 1984.

[7] G. Cousineau, P.-L. Curien, and M. Mauny. *The Categorical Abstract Machine*. 1985.

Issue VIII: Linear Interpreter

Maksym Sokhatskyi [1]

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute

1 травня 2025 р.

Анотація

Minimal language for parallel computations in symmetric monoidal categories.

Keywords: Interaction Networks, Symmetric Monoidal Categories

Присвячується автору ML

Артуру Джону Робіну Гореллу
Мілнеру

3 The Bob Language

Мова програмування **Bob** — це внутрішня мова симетричних моноїдальних категорій, що реалізує паралельні обчислення через взаємодію комбінаторів (ζ, δ, ϵ) з правилами анігіляції та комутації, придатна для моделювання лінійних і паралельних систем.

Definition 1. Терми **Bob** складаються зі змінних, комбінаторів (Con, Dup, Era), пар, обміну (Swap), зв'язування (Let) та одиниці (Unit). Мова підтримує афінну логіку, забороняючи повторне використання змінних.

```
I = #identifier
BOB = I  | Con BOB   | Dup BOB | Era BOB
         | Pair (BOB, BOB)     | Unit
         | Let (I, BOB, BOB)   | Swap BOB
```

Definition 2. Кодування термів у мові OCaml:

```
type term =
    | Var of string
    | Con of term
    | Dup of term
    | Era of term
    | Pair of term * term
    | Swap of term
    | Let of string * term * term
    | Unit
```

Theorem 1. Правила обчислень у **Bob** базуються на анігіляції та комутації комбінаторів:

```
Con (Con x) → Pair (x, x)
Dup (Dup x) → Pair (x, x)
Era (Era x) → Unit
Con (Dup x) → Dup (Con x)
Con (Era x) → Pair (Era x, Era x)
Dup (Era x) → Pair (Era x, Era x)
Swap (Pair (t, u)) → Pair (u, t)
Let (x, t, u) → subst x t u
```

$$\frac{\zeta(\zeta(x))}{(x,x)} \qquad (\zeta\text{-annihilation})$$

$$\frac{\delta(\delta(x))}{(x,x)} \qquad (\delta\text{-annihilation})$$

$$\frac{\varepsilon(\varepsilon(x))}{\mathbf{1}} \qquad (\epsilon\text{-annihilation})$$

Definition 3. Підстановка в **Bob**:

```
let rec subst env var term = function
    | Var v ->
        if v = var then
            if is_bound var env then failwith "Affine violation: variable used twice"
            else term
        else Var v
    | Con t -> Con (subst env var term t)
    | Dup t -> Dup (subst env var term t)
    | Era t -> Era (subst env var term t)
    | Pair (t, u) -> Pair (subst env var term t, subst env var term u)
    | Swap t -> Swap (subst env var term t)
    | Let (x, t1, t2) ->
        let t1' = subst env var term t1 in
        if x = var then Let (x, t1', t2)
        else Let (x, t1', subst env var term t2)
    | Unit -> Unit
```

Definition 4. Редукція термів у **Bob**:

```
let reduce env term =
    match term with
    | Con (Con x) -> Pair (x, x)
    | Dup (Dup x) -> Pair (x, x)
    | Era (Era x) -> Unit
    | Con (Dup x) -> Dup (Con x)
    | Con (Era x) -> Pair (Era x, Era x)
    | Dup (Era x) -> Pair (Era x, Era x)
    | Swap (Pair (t, u)) -> Pair (u, t)
    | Let (x, t, u) -> subst env x t u
    | _ -> term
```

Definition 5. Пошук активних пар для редукції:

```
let rec find_redexes env term acc =
    match term with
    | Con (Con x) -> (term, Pair (x, x)) :: acc
    | Dup (Dup x) -> (term, Pair (x, x)) :: acc
    | Era (Era x) -> (term, Unit) :: acc
    | Con (Dup x) -> (term, Dup (Con x)) :: acc
    | Con (Era x) -> (term, Pair (Era x, Era x)) :: acc
    | Dup (Era x) -> (term, Pair (Era x, Era x)) :: acc
    | Swap (Pair (t, u)) -> (term, Pair (u, t)) :: acc
    | Let (x, t, u) -> (term, subst env x t u) :: find_redexes env t (find_redexes env u acc)
    | Con t ->
        (match t with
         | Dup _ | Era _ -> acc
         | Con x -> find_redexes env t ((term, reduce env term) :: acc)
         | _ -> find_redexes env t acc)
    | Dup t -> find_redexes env t acc
    | Era t -> find_redexes env t acc
    | Pair (t, u) ->
        let acc' = find_redexes env t acc in
```

```
        find_redexes env u acc'
    | Swap t -> find_redexes env t acc
    | Var _ | Unit -> acc
```

Definition 6. Паралельна редукція:

```
let eval_parallel pool env term =
    let rec loop term =
        let redexes = find_redexes env term [] in
        if redexes = [] then term
        else
            let new_term = Task.run pool (fun () ->
                List.fold_left
                    (fun acc (old_t, new_t) -> replace_subterm old_t new_t acc)
                    term redexes
            ) in
            loop new_term
    in
    loop term
```

Theorem 2. Доведення, що мова **Bob** є внутрішньою мовою симетричних моноїдальних категорій:

$$\begin{cases} \text{Let} : A \to C(u \cdot t, t : A \to B, u : B \to C), \\ \text{Pair} : A \to B \to A \otimes B, \\ \text{Swap} : A \otimes B \to B \otimes A, \\ \text{Con} : A \otimes A \to A, \\ \text{Dup} : A \to A \otimes A, \\ \text{Era} : A \to \mathbf{1}, \\ \text{Var} : A, \\ \text{Unit} : \mathbf{1}. \end{cases}$$

Конструктори відповідають аксіомам СМК:

- Pair моделює тензорний добуток $\otimes$. - Swap реалізує симетрію $\sigma_{A,B}$ з умовою $\sigma_{B,A} \circ \sigma_{A,B} = \text{id}_{A \otimes B}$. - Unit є одиничним об'єктом I для якого $A \otimes I \cong A$. - Let моделює композицію морфізмів (асоціативність). - Dup та Era утворюють структуру комоноїда. - Con діє як контракція.

Лямбда-функція і аплікація:

$$\begin{cases} \lambda x.t \vdash \text{Con}(\text{Let}(x, \text{Var}(x), t)), \\ tu \mapsto \text{Con}(\text{Pair}(t, u)). \end{cases}$$

Issue IX: Quantum Interpreter

Максим Сохацький [1]

[1] Національний технічний університет України
«Київський Політехнічний Інститут» ім. Ігора Сікорського
29 жовтня 2018

Анотація

Ця робота є спробою огляду існуючих мов програмування для квантових обчислень, їх теоретичних основ та особливостей. Як приклад розглядається дискретне перетворення Фур'є, яке в якості вправи імплементується на двох мовах програмування: практичній, імперативній QCL від Бернхарда Омера[12] та теоретичній формальній, функціональній мові програмування, лямбда-численні з лінійними функціями від Андре ван Тондера[16]. Як висновок пропонуємо нарис своєї мови PLQ для квантових обчислень з залежними, залежними-лінійними та квантовими типами.
Ключові слова: Теорія типів, квантові комп'ютери

Присвячується

Феліксу Блоху

4 The Felix Language

4.1 Лінійна алгебра

Нотація Дірака це компактний формалізм лінійної алгебри який будемо застосовувати для визначень квантової механіки[1].

Табл. 3: Нотація Дірака

Нотація	Визначення
$\|\psi\rangle$	загальний кет-вектор, наприклад $(c_0, ..., c_n)^T$
$\langle\psi\|$	дуальний бра-вектор, наприклад $(c_0^*, ..., c_n^*)$
$\|n\rangle$	n-й базис вектор стандартного базису $N = (\|0\rangle, ..., \|n\rangle)$
$\|\tilde{n}\rangle$	n-й базис вектор альтернативного базису $\tilde{N} = (\|\tilde{0}\rangle, ..., \|\tilde{n}\rangle)$
$\langle\phi\|\psi\rangle$	скалярний добуток
$\|i, j\rangle$	тензорний добуток базисних векторів $\|i\rangle$ та $\|j\rangle$
$\|\phi\rangle \otimes \|\psi\rangle$	тензорний добуток

Definition 7. (Векторний простір). Множина V називається векторни мпростором над скалярним полем F, тоді і тільки тоді, коли визначені операції $+ : V \times V \to V$ (сума векторів) та $\cdot : F \times V \to V$ (добуток скаляра та вектора) з наступними властивостями: i) $(V, +)$ утворюють комутативну групу; ii) $\lambda|\psi\rangle = |\psi\rangle\lambda$; iii) $\lambda(\mu|\psi\rangle) = (\lambda\mu)|\psi\rangle$; iv) $(\lambda + \mu)|\psi\rangle = \lambda|\psi\rangle + \mu|\psi\rangle$; v) $\lambda(|\psi\rangle + |\varphi\rangle) = \lambda|\psi\rangle + \lambda|\varphi\rangle$. Далі будемо розглядати скалярне поле комплексних чисел $F = C$.

Definition 8. (Скалярний добуток). Функція $\langle\cdot|\cdot\rangle : V \times V \to C$ називається скалярним добутком, тоді і тільки тоді, коли: i) $\langle\psi|(\lambda\varphi\rangle + \mu|\chi\rangle) = \lambda\langle\psi|\varphi\rangle + \mu\langle\psi|\varphi\rangle$; ii) $\langle\psi|\varphi\rangle = \langle\varphi|\psi\rangle^*$; iii) $0 < \langle\psi|\psi\rangle \in \mathbb{R}$. Скалярний добуток визначає норму $\| |\psi\rangle \| = \sqrt{\langle\psi|\psi\rangle} = \| \psi \|$.

Definition 9. (Повний векторний простір). Нехай V векторний простір з нормою $\| \cdot \|$ та $|\psi_n\rangle \in V$ послідовність векторів. i) $|\psi\rangle$ є послідовністю Коші ттт. $\forall\epsilon > 0 \exists N > 0 : \forall n, m > N, \| |\psi_n\rangle - |\psi_{n+1}\rangle \| < \epsilon$. ii) $|\psi\rangle$ сходиться ттт. $\forall\epsilon > 0 \exists N > 0 : \forall n > N, \| |\psi_n\rangle - |\psi\rangle \| < \epsilon$. Простір V повний ттт. кожна послідовність Коші сходиться.

Definition 10. (Гільбертів простір). Повний векторний простір H зі скалярний добутком $\langle\cdot|\cdot\rangle$ та відповідною нормою $\| \psi \| = \sqrt{\langle\psi|\psi\rangle}$ називається Гільбертовим.

[1] І.О. Вакарчук. Квантова Механіка. 2012

Definition 11. (Лінійний оператор). Нехай V – векторний простір, а A – функція $A : V \to V$. Тоді A називається лінійним оператором ттт.

$$A(\lambda|\psi\rangle + \mu|\varphi\rangle) = \lambda A|\psi\rangle + \mu A|\varphi\rangle$$

В C^n лінійний оператор є матрицею $m \times n$ з елементами $a_{i,j} = \langle i|A|j\rangle$, де $A = \Sigma_{i,j} a_{i,j}|i\rangle\langle j|$. За визначенням лінійності оператор A можно записати через лінійну суму векторів базису B:

$$A : |n\rangle \to \Sigma_k a_{kn}|k\rangle, \text{ де } |k\rangle \in B.$$

Definition 12. (Тензорний добуток гільбертових просторів). Нехай H_1 та H_2 — Гільбертові простори з базисами B_1 та B_2. Тоді тензорний добуток

$$H = H_1 \otimes H_2 = \{\Sigma_{|i\rangle \in B_1} \Sigma|j\rangle \in B_2 c_{ij}|i,j\rangle \| c_{ij} \in \mathbb{C}\}.$$

також Гільбертів простір з базисом $B = B_1 \times B_2$ та скалярним добутком:

$$\langle i,j|i',j'\rangle = \langle i|i'\rangle\langle j|j'\rangle = \delta_{ii'}\delta jj', \text{ де } |i\rangle, |i'\rangle \in B_1, |j\rangle, |j'\rangle \in B_2.$$

Definition 13. (Тензорний добуток лінійних операторів). Нехай A та B лінійні оператори на Гільбертових протосторах H_1 та H_2, тоді тензорний добуток

$$A \otimes B = \Sigma_{i,j} \Sigma_{i',j'} |i,j\rangle\langle i|A|i'\rangle\langle j|B|j'\rangle\langle i',j'|$$

лінійний оператор на на гільбертовому просторі $H_1 \otimes H_2$.

Definition 14. (Комутатор та антикомутатор). Нехай A та B лінійні оператори на гільбертовому просторі H. Оператор $[A,B] = AB - BA$ називається комутатором, а $A,B = AB + BA$ називається антикомутатором.

4.2 Інтерпретація квантової механіки

В залежності від того як саме моделюються та конструюються гільбертові простори та гамільтоніани, виникають різні теорії, від нерятивістської квантової електродинаміки до квантової хронодинаміки яка вводить поняття кварків та глюонів.

Теорія квантових обчислень — це ще одна теорія поверх абстрактного квантового формалізму та є інтерпретацією квантової механіки. Однак це не фізична теорія в тому сенсі, що вона не описує природній процес, а є ближчою до схемотехніки, з квабітами та квантовими вентилями, без визначення як саме моделюється квантова система, вона може бути або фізичним об'єктом або симулятором.

Точно так як для апаратного забезпечення будуються мови порграмування та вищі мови програмування, так само для квантових обчислень, квантових станів та квантових логічних елементів (вентилів), існують свої мови програмування. У наступній секції дамо огляд існуючих мов та підходів до їх побудови, а тут дамо основні принципи та компоненти архітекти квантових обчислень, аби пояснити основні мовні елементи, та їх перетворення.

4.3 Пам'ять квантового комп'ютера

Definition 15. (Квантовий біт). Квантовий біт або квабіт визначається як квантова система, стан якої може бути повністю виражений як суперпозиція (лінійна комбінація) двох ортонормованих власних базових станів позначених $|0\rangle$ та $|1\rangle$. Загальний стан $|\psi\rangle$ квабіта тоді визначається як $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, |\alpha|^2+|\beta|^2 = 1$. Значення квабіта описується спостереженням $N = |1\rangle\langle 1|$. $\langle N\rangle$ дає вірогідність знайти систему в стані $|1\rangle$, якщо над квабітом були проведені виміри. Простір станів квабіта є гільбертовим простором $H = \mathbb{C}^2$. Ортонормована система $|0\rangle, |1\rangle$ називається обчислювальним базисом.

Definition 16. (Сфера Блоха). Загальний стан квабіта може бути виражений в полярних координатах θ та ϕ:

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + e^{i\phi}sin\frac{\theta}{2}|1\rangle.$$

Одиничний вектор стану $|\psi\rangle$ називається вектором Блоха $\tilde{r}_\psi$, та має наступну властивість $\tilde{r}_\phi = -\tilde{r}_\xi \leftrightarrow \langle\phi|\xi\rangle = 0$. Оберти навколо осей X, Y та Z вектора

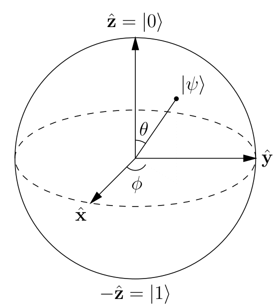

Рис. 4: Сфера Блоха як представлення квабіта $|\psi\rangle$

Блоха тоді визначаються як оператори:

$$X(\theta) = \begin{bmatrix} cos\frac{\theta}{2} & -isin\frac{\theta}{2} \\ -isin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix}, Y(\theta) = \begin{bmatrix} cos\frac{\theta}{2} & -sin\frac{\theta}{2} \\ sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix}, Z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}$$

Definition 17. (Квантова система).

Definition 18. (Еволюція системи). Темпоральна еволюція стану системи описується рівнянням Шрьодінгера:

$$i\hbar\frac{\delta}{\delta t}|\psi\rangle = H|\psi\rangle.$$

де $\hbar \equiv 1.05457{\cdot}10^{-43}$ експериментальна константа Планка, а H — фіксований самоспряжений оператор на гільбертовому просторі, знаний як Гамільтоніан квантової системи. В квантовій фізиці нормують відносно $\hbar$ тоді рівняння можна записани у безвимірній формі $i|\psi\rangle = H|\psi\rangle$. Гамільтоніан H повністю визначає квантову систему. Другий спосіб визначення, через унітарний оператор $U = e^{-iH}$. Темпоральна еволюція замкненої квантової системи зі страну $|\psi\rangle$ та часу t_1 в стан $|\psi'\rangle$ та часу t_2 може бути описана унітарним оператором $U = U(t_2 - t_1)$, таким, що $|\psi'\rangle = U|\psi\rangle$.

Definition 19. (Вимірювання). Проективне вимірювання визначається як самоспряжений оператор M який називається спостереженням зі спектральною композицією $M = \Sigma_m m P_m$, де P_m проекція на власний простір власного значення m. Власні значення m оператора M відповідаються усім можливим результатам вимірювання. Вимірювання $|\psi\rangle$ дасть результат m з вірогідністю $p(m) = \langle\psi|P_m|\psi\rangle$, таким чином через скорочення $|\psi\rangle$ отримаємо новий стан системи $|\psi'\rangle = \frac{1}{\sqrt{p(m)}} P_m|\psi\rangle$. Для стану квабіта, самоспряжений оператор N знаний як стандартне вимірювання.

$$N = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = 0 \cdot |0\rangle\langle 0| + 1 \cdot |1\rangle\langle 1|.$$

Біль загально для простору стану $H = C^n$ стандартне вимірювання визначається як $N = \Sigma_i i|i\rangle\langle i|$.

Definition 20. (Виважене середнє). Виважене середнє $\langle M\rangle$ усіх можливих результатів вимірювання M називється очікуваним значенням та визначається як

$$\langle M\rangle = \Sigma_n p(m)m = \Sigma_m \langle\psi|mP_m|\psi\rangle = \langle\psi|M|\psi\rangle.$$

4.4 Оператори обчислювального ядра

Definition 21. (Оператори Паулі). Оператори Паулі пов'язані з довільним обертом вектора Блоха формулою:

$$R_{\tilde{n}}(\theta) = e^{-\frac{i}{2}\theta\tilde{n}\cdot(x,y,z)} = cos\frac{\theta}{2}I - isin\frac{\theta}{2}(n_x X + n_y Y + n_z Z)$$

де X,Y,Z — оператори:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, Z = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

Найпростіший випадок унітарного перетворення це оператор який діє на одиничний квабіт. Загальна форма 2-вимірної комплексної унітарної матриці $U \in SU(2)$ має вигляд:

$$U = e^{i\varphi} \begin{pmatrix} e^{\frac{i}{2}(-\alpha-\beta)}cos\frac{\theta}{2} & -e^{\frac{i}{2}(-\alpha+\beta)}sin\frac{\theta}{2} \\ e^{\frac{i}{2}(\alpha-\beta)}cos\frac{\theta}{2} & e^{\frac{i}{2}(\alpha+\beta)}sin\frac{\theta}{2} \end{pmatrix}$$

Нехай $U \in SU(2)$ має базисні вектори $|u\rangle$ та $|v\rangle$ та власні значення u та v. Тоді U можна виразити:

$$U = u|u\rangle\langle u| + v|v\rangle\langle v| = e^{i\varphi}R_{\tilde{n}}(\delta),$$

де δ — фазова різниця між u та v — пов'язана як $v = e^{i\delta}v$.

Definition 22. (Оператор Адамара).

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Definition 23. (Фазовий та $\pi/8$ оператори).

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, R_3 = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$$

Definition 24. (Контрольований вентиль). Нехай U унітарний q-квабітний вентиль, контрольований U-вениил з n-контрольними бітами тоді визначається як

$$C^m[U] = \begin{bmatrix} I & \dots & 0 & 0 \\ \vdots & \ddots & 0 & 0 \\ 0 & 0 & I & 0 \\ 0 & 0 & 0 & U \end{bmatrix}$$

в просторі станів $B^{\otimes n+m}$.

Definition 25. (Контрольований НЕ вентиль).

$$CNot : |x, y\rangle \to |x \otimes y, y\rangle$$

$$CNot = C[X] = \begin{bmatrix} I & 0 \\ 0 & X \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Definition 26. (Вентиль перестановки).

$$Swap : |x, y\rangle \to |y, x\rangle$$

Definition 27. (Контрольований фазовий вентиль).

$$CPhase : |x, y\rangle \to i^{xy}|x, y\rangle$$

Definition 28. (Вентиль Тофолі).

$$CCNot : |x, y, z\rangle \to i^{xy}|x, y\rangle$$

4.5 Огляд існуючих мов

З огляду на новизну предмету розроблено не так багато мов, усі що є можна розділіти на імперативні (по духу своєї імплемантації) та функціональні (або побудовані на базі певного виду лямбда числення). Ми наведемо приклади програм для обох підходів. У якості порівняльної характеристики візьмемо алгоритм дискретного перетворення Фур'є.

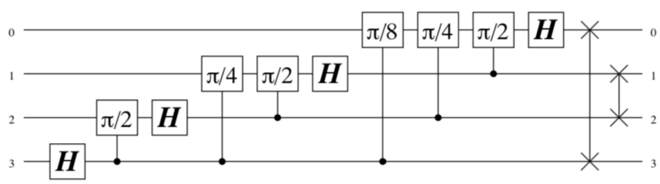

Рис. 5: Графічне представлення квантової схеми вентилів дискретного перетворення Фур'є для 4-квабітного регістру

4.5.1 Імперативні мови програмування

Станом на 2018 рік найбільш розробленою мовою, яка компілюється під Linux та Mac є QLC від Бернхарда Омера [2]. До QLC можна відзначти наступні мови: 1) Q-gol від Грега Бейкера (1996) [3]; 2) qGCL від Паоло Зуліані (2000)[14]; 3) Quantum C Language від Стефана Блаха (2002); 4) QPAlg від Марі Лолір та Філіпа Жорранда (2004)[11], синтаксис та семантика цієї системи базується на численні процесів Мілнера CCS та формальній мові LOTOS; 5) CQP (Communication Quantum Processes) від Саймона Гея та Раджагопала Нагараджана (2004)[8]; 6) Q (DSL мова для C++) від Беттеллі, Каларко та Серафіні[]. 7) LanQ від Гинека Млнаріка[13].

Definition 29. (Перестановка).

```
cond qufunct Swap(qureg a,qureg b) {
  int i; if #a!=#b { exit "arguments must equal"; }
  for i=0 to #a-1 {
    CNot(b[i],a[i]);
    CNot(a[i],b[i]);
    CNot(b[i],a[i]); } }
```

Definition 30. (Зміна порядку квабітів).

```
cond qufunct flip(qureg q) {
  int i; for i=0 to #q/2-1 { Swap(q[i],q[#q-i-1]); } }
```

Definition 31. (Дискретне перетворення Фур'є, Коперсміт).

```
operator dft(qureg q) {
  const n=#q; int i; int j;
  for i=1 to n {
      for j=1 to i-1 { V(pi/2^(i-j),q[n-i] & q[n-j]); }
      H(q[n-i]); }
  flip(q); }
```

[2] Bernhard Ömer. Structured Quantum Programming. PhD. TU Vienna. 2003. `http://tph.tuwien.ac.at/~oemer/doc/structquprog.pdf`

[3] Gregory David Baker. Qgol. A system for simulating quantum computations: Theory, Implementation and Insights. 1996. PhD. Macquarie University.
`http://www.ifost.org.au/~gregb/q-gol/QgolThesis.pdf`

4.5.2 Функціональні мови програмування

Лямбда числення лежить в основі сучасних алгебраїчних систем та основ математики[6], воно може бути розглянуте не лише як мова програмування загального використання, але і фреймворк для судження про обчислювальні властивості програм.

Таксономію лямбда числень, яку виконав Хенк Барендрегт та подав у вигляді лямбда кубу[7], можна розширити квантовими мовними примітивами, вводити в вищі лямбда числення з гомотопічними типами, або в системи доведення теорем з екстрактом або лише верифікацією.

З огляду на природу квантових обчислень необхідним компонентом квантового лямбда числення є система лінійних типів, де за час існування змінної в області видимості під час виконання дозволяється звертання до неї тільки один раз. Така система типів уже використовується не тільки в еспериментальних верифікаторах але і в сучасних системних мовах програмування (Rust), де завдяки верифікатору лініних типів вдається обійтися без алгоритмів автоматичного вивільнення памяті під час виконання програми (схема памяті повністю моделюється під час компіляції програми).

Серед робіт присвячених мовам на базі лямбда числень можна відзначити наступні: 1) Найбільш класичне нетипизоване лямбда числення Андре ван Тондера[16]. Тут подається доведення ізоморфізму машині Тюрінга, повнота та звучання системи, що є формальною перевагою перед імперативними мовами, такими як QCL; 2) Quipper від Гріна, Люмсдейна, Росса, Селінжера та Валірона [10][9]. Це спроба вбудувати DSL для квантових обчислень в мову Haskell, симулятор мови побудований на генераторах групи Кліффорда, квантових операторах X,Y,Z,H,S. 3) Робота по лямбда численню від Аррігі та Довека[5]. 4) QML для Haskell від Торстена Альтенкірха та Джонатана Граттажа[1][2][3].

Definition 32. (Синтаксичне дерево O_H). Синтаксичне дерево O_H визначає лямбда числення з лінійними змінними поєднане з класичним нетипизованим лямбда численням. Тобто O_H є найпростішим операційним квантовим лямбда численням для середовищ виконання.

$$
\begin{aligned}
c &= \mathbf{0} \mid \mathbf{1} \mid \mathbf{H} \mid \mathrm{S} \mid R_3 \mid \mathbf{CNot} \mid \mathrm{X} \mid \mathrm{Y} \mid \mathrm{Z} \mid \ldots \\
t &= x \mid \lambda\, x\, .\, t \mid \mathbf{let}\ x = t\ \mathbf{in}\ t \mid t\ t \\
&\quad \mid (t, t) \mid t \mid c \mid !\, t \mid \lambda\, !\, x\, .\, t
\end{aligned}
$$

Тут даються примітиви лінійних типів, де доступ до змінної можливий лише раз в обсласті визначення змінної. Для нелінійних або звичайних лямбда функції даються примітиви позначені !. В переліку квантових примітивів c даються: i) ортонормований базис $|0\rangle$ та $|1\rangle$; ii) H — оператор Адамара; iii) S — фазовий вентиль; iv) R_3 — $\pi/8$ вентиль; v) контрольований не вентиль CNot; vi) вентилі Паулі X, Y та Z.

Definition 33. (Пара Айнштайна-Подольського-Розена).

$$\mathbf{EPR} = \mathbf{CNot}\ ((\mathbf{H}\ \mathbf{0}), \mathbf{0}) \tag{1}$$

Definition 34. (Квантова телепортація).

$$
\begin{aligned}
\mathbf{teleport}\ x = \mathbf{let}\ (e_1, e_2)\ &=\ \mathbf{EPR}\ \mathbf{in} \\
&\mathbf{let}\ (x', y') = \mathbf{alice}\ (x, e_1)\ \mathbf{in}\ \mathbf{bob}\ (x', y', e_2)
\end{aligned}
\tag{2}
$$

$$
\begin{aligned}
&\mathbf{alice}\ (x, e_1) = \mathbf{let}\ (x', y') = \\
&\qquad\qquad \mathbf{CNot}\ (x, e_1)\ \mathbf{in}\ ((\mathbf{H}\ x'), y') \\
&\qquad \mathbf{bob}\ (x', y', e_2) = \mathbf{let}\ (y'', e_2') = cX\ (y', e_2)\ \mathbf{in} \\
&\qquad\qquad\qquad \mathbf{let}\ (x'', e_2'') = cZ\ (x', e_2')\ \mathbf{in}\ (x'', y'', e_2'')
\end{aligned}
\tag{3}
$$

Definition 35. (Дискретне перетворення Фур'є).

$$\mathbf{fourier}\ \mathit{list}\ =\ \mathbf{reverse}\ \mathbf{fourier}'\ \mathit{list} \tag{4}$$

$$
\mathbf{fourier}'\ \mathit{list}\ =\ \mathbf{case}\ \mathit{list}\ \mathbf{of}
\begin{cases}
() \rightarrow () \\
h : t \rightarrow \mathbf{let}\ h'\ :\ t' = \mathbf{phases}\ (\mathbf{H}\ h)\ t\ !2 \\
\mathbf{in}\ h'\ :\ \mathbf{fourier}'\ t'
\end{cases}
\tag{5}
$$

$$
\begin{aligned}
&\mathbf{phases}\ \mathit{target}\ \mathit{controls}\ !n\ = \\
&\quad \mathbf{case}\ \mathit{control}\ \mathbf{of}
\begin{cases}
() \rightarrow \mathit{target} \\
\mathit{control} : t \rightarrow \mathbf{let}\ (\mathit{control}', \mathit{target}') = \\
(cR\ !n)\ (\mathit{control}, \mathit{target})\ \mathbf{in} \\
\mathbf{let}\ \mathit{target}'' : t' = \mathbf{phases}\ \mathit{target}'\ t\ !(\mathbf{succ}\ n)\ \mathbf{in} \\
\mathit{target}'' : \mathit{control}' : t'
\end{cases}
\end{aligned}
\tag{6}
$$

4.6 Висновки

Як видно для реалізації семантики мови програмування для квантових комп'ютерів достатньо поєднаяти тензорне числення разом з π-численням процесів, або лінійними типами. На сьогоднішній день (2018) серед імперативних мов програмування найбільш завершена, повна та практична на думку автора є QLC від Бернхарда Омера, серед мов для лямбда числень немає жодної достатньо зрілої імплементації.

4.7 Мова PLQ

Відкрите питання в теорії типів, імплементація та дизайн мови з лінійними та залежними типами та квантовими примітивами. Тому ми пропонуємо як висновок після огляду мов та їх синтаксисів свій синтаксис такої мови, та показуємо з яких інгрідієнтів її можна побудувати.

Мову квантових обчислень O_H можна розкласти до комбінації (прямої суми) трьох синтаксичних дерев: i) дерева O_λ — базового чистого лямбда числення з залежними типами (pure, імплементовану автором[15]); ii) дерева O_π — числення з лінійними типами, або просто інший вид стрілок (linear); iii) операторний базис квантового числення O_Q з конструкторами X,Y,Z,S,H,R_3 та конструктором визначення квабіта (quantum). Тоді формальний синтаксис мови для квантових комп'ютерів, записаний на **cubicaltt**[4] у вигляді індуктивного типу синтаксичного дерева, буде виглядати так:

```
data pure (lang: U)
   = star (n: nat) | var (l: nat) | pi (l: nat) (f: lang)
   | lambda  (x: nat) (f: lang)    | app (f a: lang)

data linear (lang: U)
   = star (n: nat) | var (l: nat) | pi (l: nat) (f: lang)
   | lambda  (x: nat) (f: lang)    | app (f a: lang)

data quantum (lang: U)
   = register (n:nat) | i (n:nat) | 0 | 1 | H (l:lang) | S (:lang)
   | X (f:lang) | Y (f:lang) | Z (f:lang) | CNot (f a: lang)
```

Результуюча авторська мова (назвемо її PLQ) виражається як взаяморекурсивна сума елментарних мовних синтаксисів.

```
data PLQ =Pure      (_: pure     PLQ)
          | Linear   (_: linear   PLQ)
          | Quantum  (_: quantum PLQ)
```

З огляду на розмів реферату, семантику цієї мови наводити не будемо, однак, очевидно, що мову Андре ван Тондера можна продовжиди до PTS з лінійними типами в категоріях з сімейставами зі значеннями в симетричних моноїдальних категоріях, як було показано Матейсом Вакаром[**?**].

Література

[1] Thorsten Altenkirch and Jonathan Grattage. *A functional quantum programming language.* Logic in Computer Science, Proceedings. 20th Annual IEEE Symposium LICS '05, IEEE, 2005, pp. 249–258.

[2] Thorsten Altenkirch, Jonathan Grattage, Juliana K. Vizzotto, and Amr Sabry. *An algebra of pure quantum programming.* Electronic Notes in Theoretical Computer Science, 170:23–47, 2007.

[3] Jonathan Grattage. *An overview of QML with a concrete implementation in Haskell.* Electronic Notes in Theoretical Computer Science, Proceedings of the Joint 5th International Workshop on Quantum Physics and Logic and 4th Workshop on Developments in Computational Models, 270(1):165–174, QPL/DCM, 2011.

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

[5] Pablo Arrighi and Gilles Dowek. *Operational semantics for formal tensorial calculus.* 2004.

[6] Andrej Bauer, Steve Awodey, Matthieu Sozeau, Michael Shulman, Dan Licata, Yves Bertot, Peter Dybjer, and Nicola Gambino. *Homotopy Type Theory: Univalent Foundations of Mathematics.* 2013.

[7] H. P. Barendregt. *Lambda calculi with types.* Handbook of Logic in Computer Science (Vol. 2), Oxford University Press, New York, NY, USA, 1992, pp. 117–309.

[8] Simon J. Gay and Rajagopal Nagarajan. *Communicating quantum processes.* Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05, ACM, New York, NY, USA, 2005, pp. 145–157.

[9] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. *Quipper: a scalable quantum programming language.* ACM SIGPLAN Notices, vol. 48, ACM, 2013, pp. 333–342.

[10] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. *An introduction to quantum programming in Quipper.* International Conference on Reversible Computation, Springer, 2013, pp. 110–124.

[11] Marie Lalire and Philippe Jorrand. *A process algebraic approach to concurrent and distributed quantum computation: Operational semantics.* 2004.

[12] Bernhard Ömer. *Structured quantum programming.* 2003.

[13] Hynek Mlnarik. *Operational semantics and type soundness of quantum programming language LanQ.* 2007.

[14] J. W. Sanders and P. Zuliani. *Quantum programming.* In Proceedings of the 5th International Conference on Mathematics of Program Construction, MPC '00, Springer-Verlag, 2000.

[15] Namdak Tonpa. *The systems engineering of consistent pure language with effect type system for certified applications and higher languages.* AIP Conference Proceedings, vol. 1982, 2018.

[16] Andre van Tonder. *A lambda calculus for quantum computation.* SIAM J. Comput., 33(5):1109–1135, 2004.

[17] Matthijs Vakar. *Syntax and semantics of linear dependent types.* arXiv:1405.0033, 2014.

Issue X: The Alice Language

Maksym Sokhatskyi [1]

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute
1 травня 2025 р.

Анотація

У цій статті Формальна Тензорна Мова" або Лінійні Типи для Лінійної Алгебри" розглядаються лінійні системи типів, які є природним розширенням STLC для роботи з тензорами (структурами з лінійними алгебраїчними операціями), та розподіленим у просторі та часі програмуванням.

Основні роботи для ознайомлення з темою: Ling, Guarded Cubical, A Fibrational Framework for Substructural and Modal Logics, APL-like interpreter in Rust, Futhark, NumLin.

Keywords: Interaction Networks, Symmetric Monoidal Categories

5 The Alice Language

5.1 Вступ

5.2 Пі-числення і лінійні типи

Вперше семантика Пі-числення була представлена Мілнером разом з ML мовою, за шо він дістав премію Тюрінга (один з небагатьох хто заслужено). Якшо коротко то Пі-числення отримується з Лямбда-числення шляхом перетворення кожної змінної в нескінченний стрім.

Приклад 1: факторіал Наприклад у нас є факторіал записаний таким чином:

```
fac(0: int) -> 1
fac(x: int) -> x*fac(x -1)
```

Переписуємо його так шоб замість скалярного аргументу він споживав стрім аргументів, і результатом: Альтернативна версія на стрімах:

```
factorial(x: stream int): stream int -> result.set(x*fac(x.get()-1))
```

На відміну від попереднього факторіала, цей факторіал споживає довільну кількість аргументів і для кожного з них виштовхує в результуючий стрім результат обчислення факторіалу (використовуючи попередню функцію). Цей новий факторіал на стрімах представляє собою формалізацію нескінченного процесу який можна запустити, це процес підключиться до черги аргументів, яку буде споживати і до черги результату, куди буде виплюововути обчислення.

Приклад 2: скалярний добуток Функції можуть мати довільну кількість параметрів, всі ці параметри – це черги з яких нескінченний процес споживає повідомлення-аргументи і виплььовує їх в результуючу чергу-стрім. Наприклад лінійна функція яка обчислює скалярний добуток трьохвимірних векторів виглядатиме так:

```
dot3D(x: stream int, y: stream int): stream int ->
[x1, x2, x3] = x.get(3)
[y1, y2, y3] = y.get(3)
result.set(x1y1+x2y2+x2*y3)
```

Слід розрізняти лінійність як алгебраїне поняття і лінійнісь в Пі-численні. В Пі-численні, а також в лінійній логіці Жана-Іва Жирара лінійність означає шо змінна може бути використана тільки один раз, після чого курсор черги зсувається і його неможливо буде вже вернути в попередню позицію після того як якийсь процес прочитає це значення геттером. Саме така семантика присутня в цих прикладах, зокрема в аксесорах get і set.

При реальних обочисленнях може статися так, що значення прочитане з черги потрібно одразу двом функціям, тому природньо надати можливість закешувати це значення, або іншими словами створити його копію x.duplicate для передачі по мережі далі іншим функціям-процесам. Так само варто приділити увагу деструктору пам'яті коли це значення вже використане усіма учасниками і більше не потрібно нікому x.free.

5.3 BLAS примітиви в ядрі

Для реальних промислових обчислень скалярні добутки не рахують руками, а є примітивами високооптимізованих бібліотек за допомогою SPIRAL чи вручну закодовні. В статті NumLin автори зосереджуються на 1-му та 3-му рівню BLAS, а це включає наступні примітиви для BLAS рівня 1: 1) Sum of vector magnitudes (Asum); 2) Scalar-vector product (Axpy); 3) Dot product (Dotp); 4) Modified Givens plane rotation of points (Rotm); 5) Vector-scalar product (Scal); 6) Index of the maximum absolute value element of a vector (Amax). Так наступні примітиви для BLAS рівня 3: 1) Computes a matrix-matrix product with general matrices (Gemm); 2) Computes a matrix-matrix product where one input matrix is symmetric (Symm); 3) Performs a symmetric rank-k update (Syrk); 4) Декомпозиція Холецького (Posv) Огляд примітивів рівня 1:

Також зауважимо шо єдиними типами даних які є в BLAS це Int і Float, а також нам знадобляться хелпери типу Transpose і Size. Тому синтаксичне дерево вбудованих примітивів BLAS буде виглядати так:</p>

```
data Arith = Add | Sub | Mul | Div | Eq | Lt | Gt
data Builtin
   = Intop (a: Arith) | Floatop (a: Arith)      -- SIMD types
   | Get | Set | Duplicate | Free               -- linearity
   | Transpose | Size                           -- matrices
   | Asum | Axpy | Dotp | Rotm | Scal | Amax    -- BLAS Level 1
   | Symm | Gemm | Syrk | Posv                  -- BLAS Level 3
```

5.4 Лінійне лямбда числення

Лінійне лямбда числення має всього три контексти: 1) Часткових дозволів, 2) Контекст лінійних змінних, 3) контекст звичайного лямбда числення.

Як мною було показано в QPL ми можемо одночасно мати два лямбда числення: стандартне і лінійне на стрімах, однак тут ми просто будуємо звичайне лінійне лямбда числення виділяючи його з основного дерева. Тензори в пам'яті містять додаткову інформацію про часткові дозволи Fraction, якшо Fraction = 1 то мається на увазі повний ownership, якшо Fraction = 1/2 то частковий, шо означає шо дві частин програми мають доступ до нього, часткові дозволеності можна об'єднувати в процесі нормалізації аж до повного ownership (Fraction = 1). Тут Pair представляє собою лінійну пару, Fun — лінійну функцію, Consume — споживання змінної перенос її з лінійного контексту в звичайний.

```
data Fraction = Z | S (_: Fraction)
data Dimension = Vector | Matrix | Stream | Table
data Linear
   = Empty | Unit | Bool
   | Int | Float
   | Tensor (a: Fraction) (x: Dimension)
   | Pair (a b: Linear) | Fun (a b: Linear)
   | Consume (a: Linear) | All (a: Var) (b: Linear)
```

Приклад 3: лінійна регресія

$$Posv : matrix \multimap matrix \multimap matrix \otimes matrix \beta = (X^T X)^{-1} X^T y$$

Програма, яка обчислює лінійну регресію спочатку визначає розмір матриці x, потім створює в пам'яті нову матриці $X^T y$ і $x^T x$, після чого обчислює за допомогою $Posv$ і цих двох матриць безпосередньо результат.

```
Linear_Regression(x y: matrix float) ->
(n, m) = Size x
xy = Tensor (m, 1) { Transpose(x) * y }
xTx = Tensor (m, m) { Transpose(x) * x }
(w, cholesky) = Posv xTx xy
Free w
result.emit(cholesky)
```

5.5 AST результуючої мови

Повне дерево виразів:

```
data Exp
= Variable (: Var)
| Prim (: Builtin)
| Star | True | False
| Int (: nat) | Float (: float)
| Lambda (a: Var) (b: Linear) (c: Exp)
| App (a b: Exp) | Pair (a b: Var) (c d: Exp)
| Consume (a: Var) (b c: Exp)
| Gen (a: Var) (b: Exp) | Spec (a: Exp) (b: Fraction)
| Fix (a b: Var) (c d: Linear) (e: Exp)
| If (a b c: Exp) | Let (a: Var) (b c: Exp)
```

```
SimpleConvolution1D (i: int) (n : int) (x0: float)
(write: vector float) (weights: vector float): vector float ->
if n = i then result.emit(write)
a = [w0,w1,w2] = weights.get(0,3)
b = [x0,x1,x2] = [ x0 | write.get(i,2) ]
write.set(i, Dotp a b)
SimpleConvolution1D (i + 1) n x1 write weights

test ->
write   = [10, 50, 60, 10, 20, 30, 40]
weights = [1/3, 1/3, 1/3, 1/3, 1/3, 1/3, 1/3]
cnn = SimpleConvolution1D 0 6 10 write weights
[10.0, 40.0, 40.0, 30.0, 19.999999999999996, 30.0, 40.0] = cnn
```

```
write   = [10.0, 50.0, 60.0, 10.0, 20.0, 30.0, 40.0]
weights = [1/3, 1/3, 1/3, 1/3, 1/3, 1/3, 1/3]
result  = CNN.conv1D(1,6,10.0,write,weights)

initial: [10.0,50.0,60.0,10.0,20.0,30.0,40.0]
result: [10.0,40.0,40.0,30.0,19.999999999999996,30.0,40.0]
```