# Issue X: The Robin Language

Maksym Sokhatskyi [1]

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

2 червня 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

# Зміст

Присвячується автору ML

Робіну Мілнеру

# 1 The Robin Language

## 1.1 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.

## 1.2 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
```

## 1.3 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)
```

## 1.4 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]
```