Розподілене конкурентне програмування за допомогою Cloud Haskell

Микита Юрченко

Складено 26 листопада 2017 р.

Зміст

1 Вступ

Cloud Haskell - це EDSL (Embedded Domain-Specific Language) для розробки розподілених відказостійких конкурентних[1] додатків, заснований на моделі обміну повідомленнями, подібний Erlang OTP [3][4] та MPI.

Сучасні веб-додатки мають обробляти великі обсяги даних, тож виникає потреба не тільки в ефективному розподілені роботи між обчислювачами одного комп'ютера, а й між багатьма комп'ютерами, об'єднаними в кластер. Надвеликі обчислювальні кластери, які продають свої ресурси користувачам, дістали назву Хмар (Cloud).

Традиційні підходи до конкурентного програмування на багатоядерних процесорах використовують спільну пам'ять. На відміну від них, модель акторів, яку імплементовано в Ерланзі, не дозволяє жодної імпліцитної комунікації між процесами. *Абстрагування транспортного рівня від моделі виконання практично стирає принципову різницю між локальними та віддаленими обчислювачами.* Можна розробляти програму, використовуючи лише один комп'ютер, а потім, доклавши незначни зусилль, необмежено масштабувати її. Цей підхід Д. Армстронг назвав у своїй дисертації «Abstracting out concurrency»[5]. Таким чином, хоча цей підхід цілком сприятливий для програмування одного комп'ютера, його справжня сила розкривається при роботі з розподіленими веб-серверами та кластерами для наукових обчислень.

Сфера застосування Cloud Haskell дещо ширша, ніж в Ерлангу. Найбільш ефективно буде застосовувати CH для розробки алгоритмів, які передбачають великі незалежні обчислювання, наприклад:

- Великі дані
- Машинне навчання (нейронні мережі)
- Високонавантажені веб-додатки
- Відказостійкі розподілені системи
- Наукові обчислення

Cloud Haskell недоцільно використовувати для наступних задач:

[1]Традиційно прийнято перекладати concurrency як багатопоточне, або паралельне програмування. З точки зору Гаскеля різниця між concurrent та parallel є важливою[1]. Я вслід за В. Брагілевським перекладаю concurrency як конкурентність[2].

- Додатки, які потребують сильної оптимізації під конкретне залізо
- Додатки, які активно взаємодіють з іншими мовами програмування
- Тісно зчеплені (tight coupling) підзадачі, які вимагають багато синхронізації

Деякі переваги Cloud Haskell порівняно з Erlang:

- Більш продуманий синтаксис
- Кращі бібліотеки
- Дозволяє користуватися роздільною пам'яттю на одній машині
- Типізовані канали
- Система типів
- Метапрограмування
- Користувацька серіалізація

На сьогодні в багатьох популярних мовах програмування модель акторів представлено у вигляді портованої бібліотеки Scala Akka. Порівняння трьох підходів наведено у таблиці 1.

	Erlang OTP	Scala Akka	Cloud Haskell
Реліз	1987	2009	2012
Версія	20.1	2.5.6	0.7.7
Типізація	Динамічна	Дин. примітиви	Повністю статична
Гаряча заміна	Так	Так	Ні
Компіляція	BEAM VM	JVM	Нативний код
Серіалізація	Автоматична	Автоматична	Визначається користувачем (або виводиться)
Використання примітивів синхронізації	Неможливо	STM	MVar, TVar

Табл. 1: Порівняння Erlang, Scala та Haskell

2 Інсталяція

> My setup = A compiler, emacs, bash, make, an os (I'm not religious here) pencil, paper, coffee, quiet environment, good nights sleep.
>
> Джо Армстронг

Реферат було виконано з використанням ОС NixOS (GNU Linux) [6]. Стек розробки, який використовує автор, є розповсюдженим серед програмістів на Гаскелі, але програмістам на мейнстрімових мовах може здатись незвичним.

- GNU Emacs 25 (з плагіном haskell-mode)
- Haskell Stack
- NixOS 17.03
- GHC 8.02

Засіб є досить високорівневим і не накладає специфічних вимог на апаратне забеспечення. Насправді, його можна підняти як на Raspberry Pi [7], так й на суперкомп'ютері Altamira [8]. Приклади складено автором на його домашньому AMD Phenom II X4 945 3.0 GHz з 4 Gb RAM. У разі потреби, конфігураційні файли автора можна знайти у [9]

На сьогодні, єдине, що необхідно для початку роботи з Гаскелем - це Stack.

```
[nik@nixos:~/projects]$ nix-env -iA nixos.haskellPackages.stack
```

Готовий проект можно завантажити за посиланням [10]. Кроки, необхідні для самостійного відтворення, буде наведено нижче.

```
[nik@nixos:~/projects]$ stack new CloudRef simple
```

Буде створено проект з такою стуктурою:

```
[nik@nixos:~/projects]$ tree CloudRef --charset=ascii
CloudRef
|-- CloudRef.cabal
|-- LICENSE
|-- README.md
|-- Setup.hs
```

```
|-- src
|   `-- Main.hs
`-- stack.yaml
```

Лістинг 1: Структура початкового проекту

Збудуймо цей проект.

```
[nik@nixos:~/projects/CloudRef]$ stack init
[nik@nixos:~/projects/CloudRef]$ stack build
[nik@nixos:~/projects/CloudRef]$ stack exec CloudRef
```

Лістинг 2: Будування та запуск проекту

Має вивести до консолі `"hello, world"`. Тепер маємо завантажити необхідні бібліотеки. Налаштуємо відповідні рядки у файлах `stack.yaml` та `CloudRef.cabal` згідно до приведених нижче лістингів, та перебудуємо проект за допомогою `$ stack build`.

```
...
extra-deps:
- distributed-process-0.7.3
- network-transport-tcp-0.5.1
...
nix:
  enable: true
```

Лістинг 3: stack.yaml

Скорегуємо `CloudRef.cabal` як показано в наступному лістингу:

```
...
executable HelloCloud
  hs-source-dirs:      src
  main-is:             HelloCloud.hs
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5
                     , distributed-process == 0.7.3
                     , network-transport == 0.4.4.0
                     , network-transport-inmemory == 0.5.2
```

Лістинг 4: CloudRef.cabal

Створимо `src/HelloCloud.hs`, який просто виводить до консолі "Hello, Haskell!".

```
import Control.Distributed.Process
import Control.Distributed.Process.Node
import Control.Concurrent
import Network.Transport
import Network.Transport.InMemory

main :: IO ()
main = do
  -- Створимо новий транспорт
  t <- liftIO createTransport

  -- Створимо нову локальну ноду
  n <- newLocalNode t initRemoteTable

  -- Запустимо на ній процес
  forkProcess n $ do
    say "Hello, Haskell!"

  -- Чекаємо секунду, аби процес встиг
  liftIO (threadDelay (10^6))

  -- Акуратно закриваємо транспорт
  closeTransport t
```

Лістинг 5: src/HelloCloud.hs

Виконаємо наш приклад за допомогою `$ stack build && stack exec HelloCloud`. Має вивести до консолі щось, подібне до

```
...
[nik@nixos:~/projects/CloudRef]$ stack build && stack exec HelloCloud
Sat Nov 11 14:16:49 UTC 2017 pid://1:8: Hello, Haskell!
```

Лістинг 6: Будування та запуск HelloCloud.hs

Кожного разу перебудовувати цілий проект, а потім шукати рядки з помилками досить незручно. Автор радить усім користуватися плагіном haskell-mode[11] для текстового редактору Emacs[12], який можна просто завантажити за допомогою репозиторію MELPA прямо з самого Emacs[13].

Відкрийте необхідний .hs файл та виконайте `C-c C-l`. Має відкритися буфер інтерпретатору GHCi[14]. З нього ви можете запускати функції, визначені

та імпортовані в поточному файлі. щоб перезавантажити файл, достатньо повторити комбінацію `C-c C-l`, або набравши `:r` в інтерпретаторі.

Насправді, ви можете користуватися інтерпретатором й без Emacs, прямо з консолі за допомогою команди `$ stack ghci`.

Для профілювання нам стане у нагоді програма ThreadScope. Для її встановлення скористуємося `$ nix-env -iA nixos.haskellPackages.threadscope`.

Знімок робочого оточення автора наведено на рис.1.

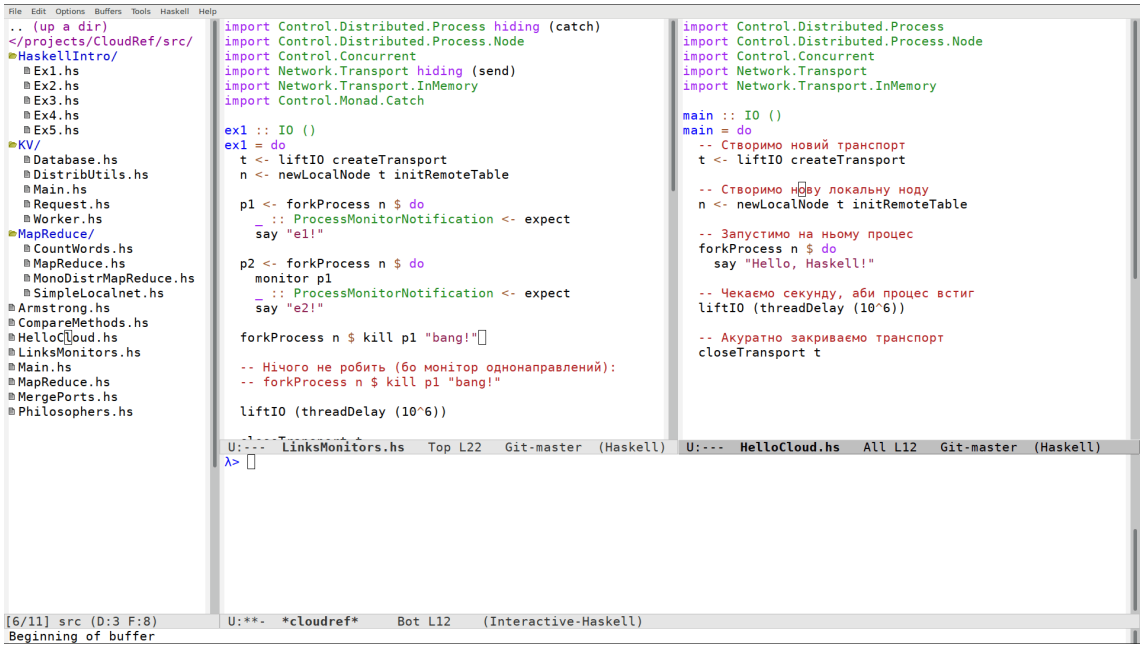


Рис. 1: Робоче оточення

Якщо все вийшло, то ми можемо вважати себе повністю готовими до занурення у світ Cloud Haskell.

3 Синтаксис та основні конструкції мови Гаскель

> My prediction is, that in the end, parallel programming will mean functional programming
>
> Саймон Пейтон-Джонс

Гаскель - поліморфна компільована чисто функціональна лінива мова програмування з багатою системою типів.

Головні властивості:

1. Вивід типів в стилі Дамаса-Гіндлі-Мілнера
2. Поліморфізм вищих рангів
3. Нестрога семантика, імплементована важким рантаймом на базі STG машини
4. Алгебраїчні типи даних

В цьому рефераті під Haskell ми маємо на увазі GHC Haskell, тобто реалізацію специфікації компілятором GHC (Glorious Glasgow Haskell Compiler).

На відміну від Java або Rust, абсолютно неможливо людину, яка не має жодного досвіду програмування на ML-подібній мові, навчити писати на достатньому для Cloud Haskell рівні, просто виклавши «деякі особливості синтаксису», або «основні інтуїції»[2].

3.1 Типи та функції

Всі приклади в цьому розділі не призначені для компіляції. Передбачається, що читач буде грати з ними з GHCi. Надалі рекомендується виконувати перевірку типів та невеликі тести з консолі GHCi, бенчмарки - лише після повноцінної компіляції.

Програмування на Haskell полягає в визначенні типів даних та функцій над ними. Наприклад, визначення типу натуральних чисел та їх складання:

```
{-# LANGUAGE GADTs #-}

data N where
  Z :: N
  S :: N -> N
```

[2]З цього не випливає, що haskell є важкою мовою програмування. Його синтаксис є тривіально простим (Див. розділ System FC на стор.58). А ось підходи та традиції дійсно неможливо охопити в рамках даного реферату.

```
add :: (N, N) -> N
add (a, b) = case b of
  S b' -> S (add (a, b'))
  Z -> a

five = add (S (S Z), (S (S (S Z))))
```

Лістинг 7: src/HaskellIntro/Ex1.hs

Типи визначаються як множини їх значень. Значеннями типа є т.з. конструктори даних, в нашому випадку це Z та S. Майте на увазі, що зазвичай використовують скорочений синтаксис вигляду `data N = Z | S N`. Типи даних є алгебраїчними - | можна розглядати як + (більш точно - як розмічене об'єднання множин), пробіл між аргументами конструктору - як * (точніше - як декартів добуток). Тож всі конструюються як суми добутків. Алгебраїчні типи є алгебраїчними настільки, що можна навіть брати їхню похідну. Наведемо в якості прикладу також визначення класичного АТД "Двійкове дерево": `data BTree a = E | Node a (BTree a) (BTree a)`. Одним з елементів цього типу буде, наприклад, `t1 = Node 2 (Node 1 E E) (Node 3 (Node 4 E E) E)`.

Функції визначаються як рівняння, в яких зазвичай використовують техніку зіставлення зі зразком. Ці рівняння мають багато спільного зі звичними математичними рівняннями, наприклад (N, N) можна розглядати як декартів добуток множини натуральних чисел з собою. Цей приклад є дещо надмірним - на початку зумисно приведено приклад анотації та використання синтаксичного розширення.

3.2 Класи типів

Можна розглядати класи типів як декотру аналогію інтерфейсу зі світу ООП[3]. Нехай до класу типів `Summable` входять всі типи, для яких ми визначимо операцію складання.

```
class Summable a where
  add :: a -> a -> a
```

Лістинг 8: src/HaskellIntro/Ex3.hs

Визначимо двох представників цього класу типів.

[3]До речі, в мові Idris вони так і називаються.

```
instance Summable Int where
  add a b = a + b

instance Summable [a] where
  add xs ys = xs ++ ys
```

Лістинг 9: src/HaskellIntro/Ex3.hs

Продемонструємо використання цієї конструкції. Програмістам на мейнстрімових мовах це має нагадати ad-hoc поліморфізм.

```
main = do
  print (1 `add` 2)
  print $ [1,2,3] `add` [4,5,6]
```

Лістинг 10: src/HaskellIntro/Ex3.hs

Цей приклад видасть помилку `Ambiguous type variable`. Це означає, що Гаскель не може підставити конкретний тип замість типової змінної. Річ у тім, що операція складання сама визначена у термінах класу типів `(+) :: Num a => a -> a -> a`. Гаскель підтримує перевантаження літералів. Тож маємо підказати компіляторові необхідний монотип:

```
main = do
  print ((1 :: Int) `add` 2)
  print $ [1,2,3] `add` [4,5,6]
```

Лістинг 11: src/HaskellIntro/Ex3.hs

Патерн «візьми дві a-шки, та зроби з них третю», продемонстрований вище, є досить розповсюдженим як в програмуванні, так й в алгебрі. Як й усі найважливіші класи типів, його назва прийшла зі світу абстрактної алгебри - Моноїд[4] (Monoid з операцією mappend). Інші важливі класи типів - Functor, Monad, Applicative - теж мають алгебраїчне[5] походження.

З точки зору Haskell Core (див. стор. 11), немає жодних Constraint-ів. Розглянемо два найбільш вживані функтори[6] - список та `Maybe a = Just a | Nothing`.

[4]Це неправда. Для моноїду необхідний також нейтральний елемент ϵ (mempty) та умова асоціативності, тож це лише магма або полугрупа. Але історично полугрупа не знайшла широкого застосування в Гаскелі.

[5]Точніше - теоретико-категорне

[6]Функтор можна розглядати як запаковане значення, всередину якого можна закинути

```haskell
-- * -> * - кайнд = тип типу
class Functor (f :: * -> *) where
  (<$>) :: (a -> b) -> f a -> f b

-- Правоасоціативний оператор пріоритету 6
infixr 6 <$>

instance Functor [] where
  f <$> [] = []
  f <$> (x:xs) = f x : f <$> xs

instance Functor Maybe where
  f <$> Nothing = Nothing
  f <$> Just x  = Just (f x)
```

Лістинг 12: src/HaskellIntro/Ex5.hs

Розцукровування виконується таким чином:

```haskell
data FunctorI f = FunctorI {
  fmap :: forall a b . (a -> b) -> f a -> f b }

_ListFunctorI :: FunctorI []
_ListFunctorI  = FunctorI { fmap = helper }
  where helper f []     = []
        helper f (x:xs) = f x : helper f xs

_MaybeFunctorI :: FunctorI Maybe
_MaybeFunctorI  = FunctorI { fmap = helper }
  where helper f Nothing  = Nothing
        helper f (Just x) = Just (f x)

-- λ> (+1) <$> [1..100] == fmap _ListFunctorI (+1) [1..100]
-- True
```

Лістинг 13: src/HaskellIntro/Ex5.hs

Тепер товсті стрілки замінюються звичайними, але треба передавати додатковий аргумент (реально передається словник instance-ів).

функцію та застосувати її до значення, яке там ховається. В паралельному програмуванні одним з найважливіших функторів є Футура (Async). Також, кожна монада є (аплікативним) функтором.

```
(<$$>) :: Functor f => (a -> a) -> f a -> f a
f <$$> box = f <$> f <$> box

fmap2 :: FunctorI f -> (a -> a) -> f a -> f a
fmap2 i f x = fmap' f (fmap' f x)
  where fmap' = fmap i

-- λ> (+1) <$$> [1..100] == fmap2 _ListFunctorI (+1) [1..100]
-- True
```

Лістинг 14: src/HaskellIntro/Ex5.hs

Більш детально про компіляцію класів типів можна почитати у[15].

3.3 Монади

> Chaque monade exprime donc le monde entier, mais obscurément, confusément, puisqu'elle est finie, et le monde, infini.
>
> Жиль Дельоз

Користуючись теорією категорій, визначити монаду тривіально просто (моноїд в категорії ендофункторів над декартово замкненою категорією Hask, де об'єкти - типи, морфізми - функції), але з очевидних причин нам доведеться обмежитися інтуїтивним описом їх практичного застосування з т.з. Haskell. Тут автор докладає до себе надлюдське зусилля, й втримується від виробництва чергового «туторіала про монади» й обмежується наведенням основних визначень стандартної бібліотеки.

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a

(>>) a b = a >>= \_ -> b
```

Лістинг 15: src/HaskellIntro/Ex4.hs

В подальшому буде наведено багато монадичного коду, записаного за допомогою do-нотації. 20 років тому, коли Haskell ще активно проектувався, постало питання про обчислення з ефектами. Пропонувалися[16] різні конструкції - засновані на продовженнях, функції, які повертають Unit, та ін. Але найбільш

цікавий варіант запропонував Пилип Вадлер - монади. З філософської точки зору[17], все, що потрапляє до монади, залишається навічно в ній. Для дотримання чистоти мови код з зовнішніми ефектами (напр, виведення до консолі) відокремлено від чистого коду шляхом загортання його до монади IO. Неможливо витягнути значення з зовнішнього світу, але можна передати туди наше обчислення. Наприклад, зчитування імені з консолі та привітання:

```
main :: IO ()
main = do
  putStrLn "Insert your name:"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

Лістинг 16: src/HaskellIntro/Ex2.hs

Зауважте, що це не є вбудованим «синтаксичним Бантустаном», в якому мова перестає бути чистою - насправді, цей код розцукровується у щось, подібне:

```
main2 = putStrLn "Insert your name:"
  >> getLine
  >>= \name ->
        putStrLn ("Hello, " ++ name ++ "!")
```

Лістинг 17: src/HaskellIntro/Ex2.hs

У спрощеному вигляді монаду IO можна представити як монотип монади State. Монада State слугує для розробки послідовних програм зі змінним станом. Має 3 основні операції: `get :: State s a`, `put :: s -> State s ()` та `evalState :: State s a -> s -> a`, тобто `type IO a = State RealWorld a`.

```
newtype State s a = State {
  runState :: s -> (a, s) }

get :: State s s
get = State $ \st -> (st, st)

put :: s -> State s ()
put newState = State $ \_ -> ((), newState)

instance Monad (State s) where
  return x = State (\s -> (x, s))
  p >>= k = q where
```

```
    p' = runState p
    k' = runState . k
    q' s0 = (y, s2) where
        (x, s1) = p' s0
        (y, s2) = k' x s1
    q = State q'

data RealWorld = RealWorld

type IO a = State RealWorld a

ex1 :: State (Int, Int) Int
ex1 =
  put (1,2)
  >> get
  >>= (\(a,b) -> return a)
```

Лістинг 18: src/HaskellIntro/Ex4.hs

В подальшому матеріал буде викладено виходячи з того, що читач знайомий з синтаксисом Гаскеля, його основними розширеннями, стандартними алгебраїчними конструкціями (напр. монадою), та має в цілому достатній рівень експертизи з Haskell.

4 Основи Cloud Haskell

4.1 Обмін повідомленнями

- `send :: Binary a => ProcessId -> a -> Process ()` - надсилає повідомлення процесові. Не блокується, не викидає жодних виключень.
- `expect :: forall a. Binary a => Process a` - заблокувати процес, доки черга повідомлень залишається пустою.
- `expectTimeout :: forall a. Binary a => Int -> Process (Maybe a)` - аналогічно `expect`, але деблокується після n мілісекунд.
- `receiveWait :: [Match b] -> Process b` - аналог ерлангівського receive. Використовується як `r <- receiveWait [ match (\m :: T -> r1), ... ]`

- `receiveTimeout :: Int -> [Match b] -> Process b` - аналог ерлангівського receive after з затримкою n мілісекунд.

4.2 Процеси та ноди, комунікація між ними.

- `spawn :: NodeId -> Closure (Process ()) -> Process ProcessId` - запускає серіалізоване замикання на вказаній ноді.
- `exit :: Binary a => ProcessId -> a -> Process ()` - делікатно (дає можливість вирішувати власну долю) завершити вказаний процес, кинувши `ProcessExitException` зі вказаною причиною.
- `kill :: ProcessId -> String -> Process ()` - насильно завершити вказаний процес.
- `catchExit :: Binary a => Process b -> (ProcessId -> a -> Process b) -> Process b` - навішує заданий гендлер відловлювання конкретного типу `ProcessExitException`-ів (другий аргумент) на заданий процес.
- `catchesExit :: Process b -> [ProcessId -> Message -> Process (Maybe b)] -> Process b` - версія `catchExit` для багатьох гендлерів одразу.
- `die :: Binary a => a -> Process b` - `exit` з поточного процесу
- `terminate :: Process a` - негайно завершити поточний процес

4.3 Лінки та монітори, обробка збоїв

- `(un)link :: ProcessId -> Process ()` - лінкування до процесу
- `(un)linkNode :: NodeId -> Process ()` - лінкування до ноди
- `(un)linkPort :: SendPort a -> Process ()` - лінкування до порту

Зауважте, що `unlink*` функції є синхронними в тому сенсі, що гарантується неможливість передачі виключення.

- `monitor :: ProcessId -> Process MonitorRef` - замоніторити процес
- `monitorNode :: NodeId -> Process MonitorRef` - замоніторити ноду
- `monitorPort :: forall a. Binary a => SendPort a -> Process MonitorRef` - замоніторити порт

- `unmonitor :: MonitorRef -> Process ()` - є асинхронною в сенсі, зазначеному вище

Якщо процес А замоніторив процес Б, то коли процес Б вмре, або від'єднається, то процес А отримає `ProcessMonitorNotification`. Можна мати довільну кількість моніторів А $\longrightarrow$ Б.

Якщо процес А прилінкувався до процесу Б, то коли процес Б вмре, або від'єднається, то в процесі А буде викинуто виключення. Неможливо мати 2 лінки А $\longrightarrow$ Б. Лінки реалізовано за допомогою моніторів

На відміну від ерлангу, монітори та лінки є однонаправленими. Розглянемо наступні 2 приклади використання моніторів та лінків відповідно.

```
ex1 :: IO ()
ex1 = do
  t <- liftIO createTransport
  n <- newLocalNode t initRemoteTable

  p1 <- forkProcess n $ do
    _ :: ProcessMonitorNotification <- expect
    say "e1!"

  p2 <- forkProcess n $ do
    monitor p1
    _ :: ProcessMonitorNotification <- expect
    say "e2!"

  forkProcess n $ kill p1 "bang!"

  -- Нічого не робить бо монітор однонаправлений:
  -- forkProcess n $ kill p2 "bang!"

  liftIO (threadDelay (10^6))

  closeTransport t

-- ghci:
-- λ> ex1
-- Wed Nov 22 13:41:54 UTC 2017 pid://1:9: e2!
```

Лістинг 19: src/LinksMonitors.hs.hs

```
ex2 :: IO ()
ex2 = do
  t <- liftIO createTransport
  n <- newLocalNode t initRemoteTable

  let h (e::SomeException) =
        liftIO $ putStrLn $ "[E] " ++ (show e)

  p1 <- forkProcess n $ handle h $ expect >>= say
  p2 <- forkProcess n $ handle h $ link p1 >> expect >>= say

  forkProcess n $ kill p1 "bang!"

  -- Не вбиває перший процес, бо лінки однонаправлені:
  -- forkProcess n $ do
  --   kill p2 "bang!"
  --   liftIO (threadDelay 1000)
  --   send p1 "I am alive!"

  liftIO (threadDelay (10^6))

  closeTransport t

-- ghci:
-- λ> ex2
-- [E] killed-by=pid://1:10,reason=bang!
-- [E] ProcessLinkException pid://1:8 DiedNormal
```

Лістинг 20: src/LinksMonitors.hs.hs

4.4 Типізовані канали

- `newChan :: Binary a => Process (SendPort a, ReceivePort a)` - створити новий канал
- `sendChan :: Binary a => SendPort a -> a -> Process ()` - надіслати повідомлення по каналу

- `receiveChan :: Binary a => ReceivePort a -> Process a` - зчитати повідомлення з каналу
- `receiveChanTimeout :: Binary a => Int -> ReceivePort a -> Process (Maybe a)` - чекати повідомлення з каналу n мікросекунд
- `mergePortsBiased :: Binary a => [ReceivePort a] -> Process (ReceivePort a)` - Об'єднує приймаючі порти, отримуючи новий порт, на який будуть приходити повідомленні я усіх переданих. При цьому, черга повідомлень перевіряється згідно списку.
- `mergePortsRR :: Binary a => [ReceivePort a] -> Process (ReceivePort a)` - аналогічно mergePortsBiased, але після кожного зчитування відбувається зсув вліво, в результаті перший порт стає останнім

Важливо відмітити таку особливість: SendPort можна передавати іншим нодам у повідомленнях, бо він є серіалізовним (тобто є екземпляром класу типів Binary, про який буде наступний підрозділ). ReceivePort є закріпленим, його не можна передавати. Це дуже важливо і зроблено навмисно для спрощення розрахунку cost model.

Проілюструємо використання цих функцій.

```
main :: IO ()
main = do
  t <- liftIO createTransport
  n <- newLocalNode t initRemoteTable

  forkProcess n $ do
    (f1, t1) <- newChan
    (f2, t2) <- newChan

    sendChan f1 "m1"
    sendChan f2 "m2"

    receiveChan t1 >>= say
    receiveChan t2 >>= say

    t3 <- mergePortsBiased [t1, t2]
    -- t3 <- mergePortsRR [t1, t2]

    sendChan f2 "m3"
```

```
  sendChan f1 "m4"
  sendChan f1 "m5"

  -- RR:     m4 -> m3 -> m5
  -- Biased: m4 -> m5 -> m3

  receiveChan t3 >>= say
  receiveChan t3 >>= say
  receiveChan t3 >>= say

liftIO (threadDelay (10^6))
closeTransport t
```

Лістинг 21: src/MergePorts.hs.hs

4.5 Клас типів Binary. Серіалізація замикань

Клас Binary a визначає 2 базові операції [7] `encode :: a -> ByteString` та `decode :: ByteString -> a` такі, що $\forall$`x.decode(encode x)` = `x`

Таким чином, користувач має сам визначити, як йому серіалізовувати той чи інший тип. Для вбудованих типів цей клас вже визначено. Реалізації автоматично наслідуються. Більш того, існує дієвий механізм автоматичного виводу екземплярів цього типу. Нехай користувач визначив тип `data T a = E | N a (T a) (T a)`. Тоді можна доповнити до `data T a = E | N a (T a) (T a) deriving (Typeable, Generic)`, а потім `instance Binary a => Binary (T a)`. Також необхідно ввімкнути розширення DeriveGeneric. В результаті, наприклад, `encode (N 'a' (N 'b' E (N 'd' E E)) (N 'c' E E)) = \SOHa\SOHb \NUL\SOHd \NUL\NUL \SOHc\NUL \NUL`. Цей приклад демонструє корисність користувацької сериіалізації - ці ж самі двійкові дерева можна було закодувати більш економно кодами Прюфера, чи якось інакше.

Виникає питання - як нам передавати функції між процесами? В Ерланзі, завдяки віртуальній машині, все є серіалізовним за замовчуванням, в тому числі й функції. Компілятор Haskell не надає подібних можливостей. Більш того, цей підхід може викликати певні складнощі. Що таке вільні змінні? Вони самі можуть бути функціями, тоді оточення цих функцій теж треба додати до

[7] Якщо подивитись вихідні коди, то там буде `class Binary t where put :: t -> Put; get :: Get t` , де `Put Get` - деякі монади. Ця більш узагальнена форма ніяк не конфліктує з нашим простим визначенням зі старої версії.

повідомлення (власне, цьому це й називається замиканням). Програміст може випадково передати величезне повідомлення лише тому, що вільною змінною деякої невеликою функції є, наприклад, база даних. В лінивій мові все стає ще важчим через наявність санків.

З цього випливає, що має сенс обмежити функції, які можна передавати. Давайте спробуємо зайти з іншого боку - які функції ми точно можемо передати? Чи можемо ми передати, наприклад, `(+)`? Звісно, що так - ця функція завжди доступна в стандартній бібліотеці. Чи можемо ми передати замкнутий терм, до якого не входить жодна вільна змінна?

Якщо один й тий самий бінарник виконується на іншому кінці, то ми можемо просто передати статичний вказівник на цей код. Оригінальна ідея, запропонована в [3], полягає в введенні на рівні компілятору конструктору типів `Static a`. Головне, що необхідно знати про статичні значення - це те, що вони є серіалізовними. Правила типізації для цих значень наведено в рис. 2, де Γ - множина зв'язок змінних та типів, для яких відомо, що вони є або не є S-зв'язаними.

Змінна є S-зв'язаною (S-bounded), якщо та тільки якщо всі вільні змінні в ній оголошено на верхньому рівні поточного модуля.

Змінна є D-зв'язаною якщо і тільки якщо вона не є S-зв'язаною.

Терм `static` e має тип `Static` τ в тому і тільки в тому випадку, коли всі вільні змінні e є S-зв'язаними.

$$\Gamma ::= \overline{x :_\delta \sigma} \quad \text{(Контекст)}$$

$$\delta ::= S | D \quad \text{(Помітка зв'язування)}$$

$$\Gamma\downarrow = \{x :_s \sigma \mid x :_s \sigma \in \Gamma\} \quad \text{(Фільтрування S-зв'язаних)}$$

$$\frac{\Gamma\downarrow \vdash e : \tau}{\Gamma \vdash \texttt{static}\ e : \texttt{Static}\ \tau} \quad \text{(Введення } \texttt{static}\text{)}$$

$$\frac{\Gamma \vdash e : \texttt{Static}\ \tau}{\Gamma \vdash \texttt{unstatic}\ e : \tau} \quad \text{(Видалення } \texttt{static}\text{)}$$

Рис. 2: Правила типізації для static

Тепер ми можемо записати `MkClosure :: Static (e -> a) -> e -> Closure a`. Але де зв'язано змінну e? Вона є екзистенційно квантифікованою, тобто можемо записати щось подібне до `[MkClosure (static (‘mod‘ 2)) 42, MkClosure (static ord) ’7’] :: [Closure Int]`. Обидві функції захоплюють різний тип e.

Спробуємо поліпшити ситуацію, додавши обмеження на тип e - він має бути серіалізовним `MkClosure :: Binary e => Static (e -> a) -> e -> Closure a`. Але навіть для такого типу не буде захоплюватися конкретний тип e, тож десеріалізація неможлива. Можна було б піти далі, передаючи якось це значення типової змінної. Але давайте згадаємо, що ми взагалі знаємо про клас Binary? Всі серіалізовні значення є взаємно відповідними до ByteString. Тож на щастя існує просте рішення - використовувати в замість e простий тип ByteString.

Для пошуку необхідних функцій генерується словник, в якому ключами є строки, а remotable функції цього модуля - значеннями. Щоб не змушувати користувача додавати кожного разу ці функції вручну, розробники Cloud Haskell скористалися вбудованим засобом метапрограмування Template Haskell. Для того, щоб помітити функцію `f` для передачі, достатньо під'єднати модуль `Control.Distributed.Process.Closure`, додати до файлу директиву компілятора `{-# LANGUAGE TemplateHaskell #-}`, та дописати після оголошення функції `$(remotable [’f])`.

Існують підходи й для серіалізації поліморфних функцій.

5 Модель акторів

> Oh, yes, this is very π-calculus stuff!
>
> Саймон Пейтон-Джонс[18]

У статті[19] побудовано математичну[8] семантику (деякої підмножини) примітивів Cloud Haskell.

5.1 Що таке актор

На жаль, на відміну від λ-числення, точне та відносно повне математичне викладення моделі акторів неможливе без важкого використання теорії доменів, операційної семантики, та ін. Насправді, це типова ситуація для практико-орієнтовних теоретичних побудувань. Тож залишається лише розважати читача

[8] Unified Semantics, запропонована Свенсоном[20]

неформальним викладом, заснованим на оглядовій статті від автора[21] та на відео[22] з його участю.

Актор має три функції.

1. Обробляє інформацію
2. Зберігає інформацію
3. Спілкується напряму з іншими акторами

Аксіоми. Коли актор отримує повідомлення, він може конкурентно:

1. надіслати повідомлення іншому акторові (в тому числі самому собі)
2. створити нових акторів
3. визначити, як він буде обробляти наступне повідомлення[9]

Порівняно з популярними імплементаціями (в т.ч. CH), оригінальна модель акторів є більш легковісною. Вона не передбачає:

1. Каналів. Всі актори спілкуються безпосередньо один з одним. Втім, канал може бути імплементовано як актор. Це відрізняє модель акторів від іншої популярної моделі процесів, що спілкуються - π-числення (Про ці відмінності буде більш детально сказано нижче).
2. Черги повідомлень (поштової скрині). Актор обробляє лише одне повідомлення. Тож відправник має самостійно смикати адресу призначення, доки не дочекається відповіді. Г'юїтт наголошує на тому, що зі стверджень «все є актором» та «кожен актор має поштову скриню» випливає, що поштова скриня теж має бути актором. В поштової скрині теж має бути поштова скриня, down to the Rabbit Hole.
3. Детермінований порядок повідомлень. Якщо актор $\mathcal{A}_1$ надсилає акторові $\mathcal{A}_2$ спочатку повідомлення m_1 а потім повідомлення m_2, то актор може отримати їх у будь-якому порядку.

Таким чином, модель акторів є, з одного боку, загальною моделлю обчисленнь (в одному ряді з λ-численням, машиною Тюрінга, продукціями Поста,

[9]Найбільш туманний пункт. Фактично, саме так актор виконує корисні обчислення. Наприклад, якщо актор "Рахунок клієнта N який зберігає стан "5 доларів отримує повідомлення "збільшити рахунок на 5 доларів то якщо наступне повідомлення спитає його про стан балансу, то він відповість "11 доларів".

рекурсивними функціями Кліні, алгорифмами Маркова, комбінаторами Шенфінкеля, та ін.), а з другого боку - моделлю паралельних обчислень (порівняну з π-численням, CCS, CSP, мережами Петрі, та ін.).

Порівняно з численнями процесів (π-численням, CCS, CSP), модель акторів не вимагає каналів (але легко може їх імплементувати). Мілнер та ін. ввели канали як примітиви, щоб отримати красиві рівняння. На практиці, якщо двоє намагаються взяти щось з виходу каналу, це вимагає досить складного механізму (two-phased commit), який суттєво збільшує оверхед.

Мережі Петрі моделюють потік управління, але не моделюють потік даних.

Згідно Г'юїтові, модель акторів не зводиться до недетермінованої машини Тюрінга. Г'юїтт особливо наголошує на незводимості індетермінізму до недетермінізму. Недетермінізм - це те, що ви отримаєте, підкинувши монетку. Індетермінізм виникає в умовах реального світу, наприклад, коли один сервер надсилає іншому два повідомлення по двох різних каналах.

5.2 Індукція по акторах

> Вот Павел Сергеевич, ну известный наш академик, Александров, он никогда не летает на самолете. А вы знаете почему? Он не знает доказательства теоремы Жуковского и поэтому не понимает, как самолет держится в воздухе. А я знаю! И не боюсь летать!
>
> Борис Миколайович Делоне

В якості ілюстрації того, що ми все ж таки маємо справу з строгим математичним формалізмом (втім не таким алгебраїчним, як π-числення), приведемо формальне визначення індукції по акторах зі статті[23] Г'юїтта та Бейкера 1977 року. Судячи з невеликого огляду автором найбільш частих посиланнь з теми, це є рідкісна *математична* стаття з теорії акторів, яка не потребує робочого знання теорії доменів, денотаційної та операційної семантики та іншої теорії рівня graduate, а використовує лише найпростішу теорію множин та порядків, відому з першого курсу будь-якому студенту-програмістові.

Визначення (Акторне обчислення)**.** *є пара* $\langle \mathcal{E}, \longrightarrow \rangle$*, де* $\mathcal{E}$ *- множина подій,* $\longrightarrow$

- строгий частковий(= транзитивний антирефлексивний) порядок на множині $\mathcal{E}$.

Визначення (Передування)**.** *Кажуть, що E_1 передує E_2 якщо $E_1 \longrightarrow E_2$. При цьому, E_2 є прямим наступником E_1.*

Визначення (Конкурентність)**.** *Кажуть, що E_1 є конкурентною з E_2, якщо невірно, що $E_1 \longrightarrow E_2$ або $E_2 \longrightarrow E_1$.*

Аксіома (Існування начальної події)**.** *Нехай для зручності множина завжди $\mathcal{E}$ має точну нижню грань $E_\perp$, яка називається начальною подією.*

Аксіома (Фінітність)**.** *Для будь-яких E_1 та E_2, множина $\{E | E_1 \longrightarrow E \longrightarrow E_2\}$ є скінченною.*

Без аксіоми фінітності індуктивні докази були б неможливі.

Визначення (Прямий попередник)**.** *E_p є прямим попередником E якщо $\nexists E' \in \mathcal{E}. E_p \longrightarrow E' \longrightarrow E$*

Аксіома (Дискретність)**.** *Для будь-якої події, множина її прямих попередників та прямих наступників є цілком визначеною.*

Наслідок. *З попередніх трьох аксіом випливає, що для будь-якого акторного обчислення існує монотонна ін'єкція у множину позитивних натуральних чисел $f : \mathcal{E} \hookrightarrow \mathbb{N}^+$*

Існування такої ін'єкції демонструє, що клас акторних обчисленнь не може розширювати клас обчислювальних функцій, запропонований Черчем.

> **Теза Черча-Тюрінга-Г'юїтта.** *Всі фізично можливі обчислювання можна напряму імплементувати за допомогою акторів.*

Дія в моделі акторів складається з об'єктів, які називаються *акторами*. Актори надсилають *повідомлення* іншим акторам, які називаються *цілями* цих повідомленнь. Отже, подія E являє собою отримання повідомлення `message(E)` актором `target(E)`.

Коли актор `target(E)` отримує повідомлення `message(E)`, він, відповідно до деякої програми обчислює новий локальний стан зі старого, та надсилає повідомлення іншим акторам. Про події, в яких отримуються ці повідомлення говорять, що вони є активованими подією E.

Визначення (Підпорядок). *Підпорядком строгого часткового порядку* $\langle A, \leq \rangle$ *називають строгий частковий порядок* $\langle A', \leq' \rangle$, *де* $\leq'$ *- транзитивне замикання підмножини* $\leq$.

Визначення (Відношення активації). *Відношення активації є деяким визначеним підпорядком відношення передування. Інтуїтивно, відношення активації схоплює поняття каузації у фізичному смислі*[10].

Наслідок. *Спостереження. Відношення активації формує дерево, коренем якого завжди є* $E_\perp$.

Визначення (Відношення прибуття). *Нехай на множині* $\mathcal{E}$ *задано відношення еквівалентності* $\simeq$ *таке, що* $E_1 \simeq E_2$ *в тому і тільки в тому випадку, коли* $\texttt{target}(E_1) = \texttt{target}(E_2)$. *Тоді кожному з отриманих класів еквівалентності задано лінейний порядок, що є підпорядком відношення передування, і називається відношенням прибуття.*

Визначення (Секція). *Секцією(перерізом) деякої впорядкованої множини* $\langle A, \leq \rangle$ *відносно деякого* $s \in A$, *є або* $\{x \in A | x \leq s\}$,*або сама ця множина.*

Наслідок. *Будь-яке відношення прибуття ізоморфно секції* $\mathbb{N}$

Теорема (Індукція по порядку передування). *Для будь-якого акторного обчислення* $\langle \mathcal{E}, \longrightarrow \rangle$ *та властивості P, якщо виконуються умови*

1. $P(E_\perp)$
2. $\forall E \in \mathcal{E}.P(\texttt{activator(E)}) \wedge P(\texttt{precursor(E)}) \implies P(E)$

то для буль-якої події $E \in \mathcal{E}$ *виконується* $P(E)$

Доказ Тривіальний за допомогою слабкої індукції за результатом монотонної ін'єкції.

Докладне та повне викладення багатьох технік формального доказу властивостей акторних программ можна знайти у[24] .

[10]Фотони - повідомлення, атоми - актори. Фотон прилітає до цілі - атому. Відбулася подія «Атом 1 отримав фотон 1». Атом переходить до збудженого стану, виділяє новий фотон 2 та повертається до стабільного стану. Про подію «Атом 2 отримав фотон 2» ми скажемо, що її було *активовано* першою подією.

6 Порівняння зі стандартними засобами

Складемо функцію `primeFactors` для розрахунку простих множників числа.

```
factorize :: Integer -> Integer -> [Integer]
factorize _ 1 = []
factorize d n
    | d * d > n = [n]
    | n `mod` d == 0 = d : factorize d (n `div` d)
    | otherwise = factorize (d + 1) n

primeFactors :: Integer -> [Integer]
primeFactors = factorize 2
```

Лістинг 22: src/CompareMethods.hs –primeFactors

Допоміжна функція для розділення списку на декілька рівновеликих підсписків:

```
divideList :: Int -> [a] -> [[a]]
divideList n xs = divideList' xs
  where m = (length xs `div` n) + 1
        divideList' [] = []
        divideList' xs = take m xs : divideList' (drop m xs)
```

Лістинг 23: src/CompareMethods.hs –divideList

В якості важкого завдання, яке можна виконувати паралельно, нехай буде знаходження суми простих множників чисел від from включно до to включно.

```
--Послідовне виконання
main0 :: IO ()
main0 = print $ sum $ map (sum . primeFactors) [from..to]
```

Лістинг 24: src/CompareMethods.hs –Sequential

```
--Вбудовані засоби конкурентного програмування
main1 :: IO ()
main1 = do
  chan <- newChan
  let works = divideList nthreads [from..to]

  forM_ works $ \xs ->
    forkIO $ do
```

```
      let work = sum $ map (sum . primeFactors) xs
      work `deepseq` writeChan chan work

  let waitThreads cnt acc = do
        if cnt == length works
          then return acc
          else do
          msg <- readChan chan
          waitThreads (cnt + 1) (msg + acc)

  print =<< waitThreads 0 0
```

Лістинг 25: src/CompareMethods.hs –Control.Concurrent

```
--Транзакційна память'
main2 :: IO ()
main2 = do
  acc <- atomically $ newTVar 0
  cnt <- atomically $ newTVar 0
  let works = divideList nthreads [from..to]

  forM_ works $
    \xs -> forkIO $ do
      let work = sum (map (sum . primeFactors) xs)
      let write = atomically $ do
            acc0 <- readTVar acc
            cnt0 <- readTVar cnt
            writeTVar acc (acc0 + work)
            writeTVar cnt (cnt0 + 1)
      work `deepseq` write

  let waitThreads = do
        cnt0 <- atomically $ readTVar cnt
        if cnt0 == length works
          then atomically $ readTVar acc
          else threadDelay 10000 >> waitThreads

  print =<< waitThreads
```

Лістинг 26: src/CompareMethods.hs –STM

```
-- Засоби Cloud Haskell
```

```
main3 :: IO ()
main3 = do
  t <- liftIO createTransport
  node <- newLocalNode t initRemoteTable
  res <- newEmptyMVar
  let works = divideList nthreads [from..to]

  let waitThreads cnt acc = do
        if cnt == length works
          then liftIO $ putMVar res acc
          else do
          msg <- expect :: Process Integer
          waitThreads (cnt + 1) (msg + acc)

  master <- forkProcess node (waitThreads 0 0)

  forM_ works $ \xs ->
    forkProcess node $ do
        let work = sum (map (sum . primeFactors) xs)
        work `deepseq` send master work

  print =<< takeMVar res
  closeTransport t
```

Лістинг 27: src/CompareMethods.hs –Cloud Haskell

Для запуску нам знадобляться такі імпорти:

```
{-# LANGUAGE BangPatterns #-}
module Main where

import System.Environment (getArgs)
import Control.Distributed.Process hiding (newChan)
import Control.Distributed.Process.Node
import Control.Concurrent
import Network.Transport hiding (send)
import Network.Transport.InMemory
import Control.Monad
import Control.Monad.STM
import Control.DeepSeq
import Control.Concurrent.STM
```

Лістинг 28: src/CompareMethods.hs –imports

Додамо новий executable до файлу CloudRef.cabal:

```
executable CompareMethods
  hs-source-dirs:      src
  main-is:             CompareMethods.hs
  default-language:    Haskell2010
  ghc-options:         -threaded
                       -eventlog
                       -rtsopts
  build-depends:       base >= 4.7 && < 5
                     , distributed-process == 0.7.3
                     , network-transport == 0.4.4.0
                     , network-transport-tcp == 0.5.1
                     , network-transport-inmemory == 0.5.2
                     , deepseq == 1.4.2.0
                     , stm == 2.4.4.1
```

Лістинг 29: CloudRef.cabal

Виміряємо наш приклад при `nthreads = 64, from = 10^9, to = from + 10^5`[11]. Для візуального аналізу роботи нашої програми будемо використовувати засіб ThreadScope. Збудуємо наш додаток `stack build` та виконаємо всі 4 варіанти. Для цього виконаємо скрипт `helper.sh`, зміст якого наведено нижче:

```
#!/bin/sh
for (( i=0; i <= 3; i++ ))
do
    echo "executing main$i"
    stack exec -- CompareMethods $i +RTS -l -N4
    cp CompareMethods.eventlog "main$i.eventlog"
done

for (( i=0; i <= 3; i++ ))
do
    threadscope "main$i.eventlog" &
done

echo "OK"
```

Лістинг 30: helper.sh

[11]Потоки в Гаскелі є легковісними, за моделлю M:N. Зазвичай найкращий приріст дає використання в 4-16 разів більше потоків, ніж ядер.

Маємо такі результати:

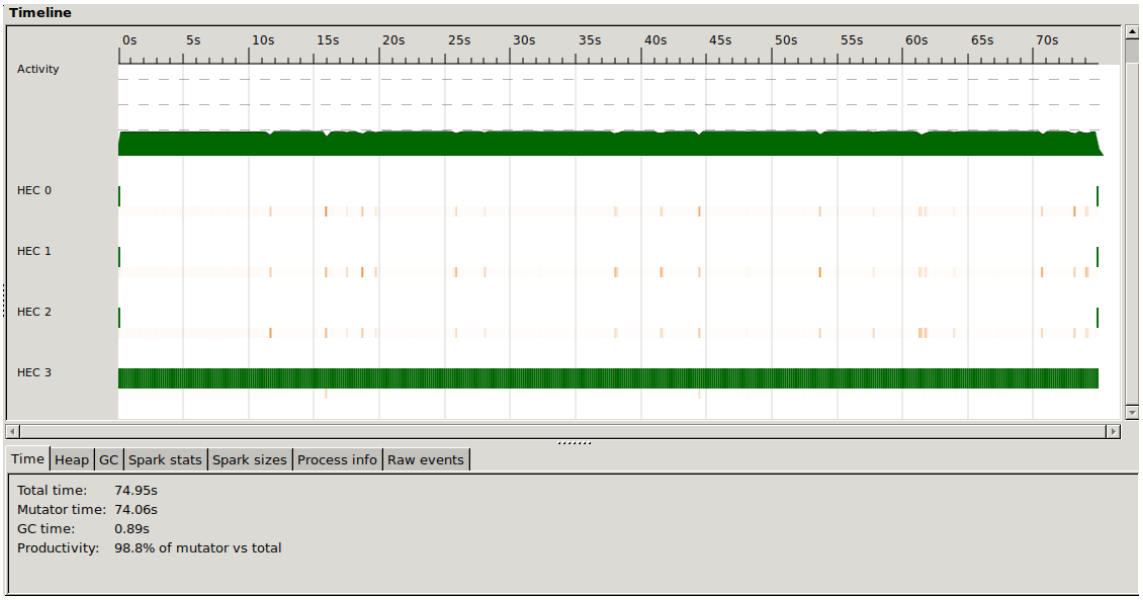


Рис. 3: Послідовне виконання за 74.95с (ThreadScope)

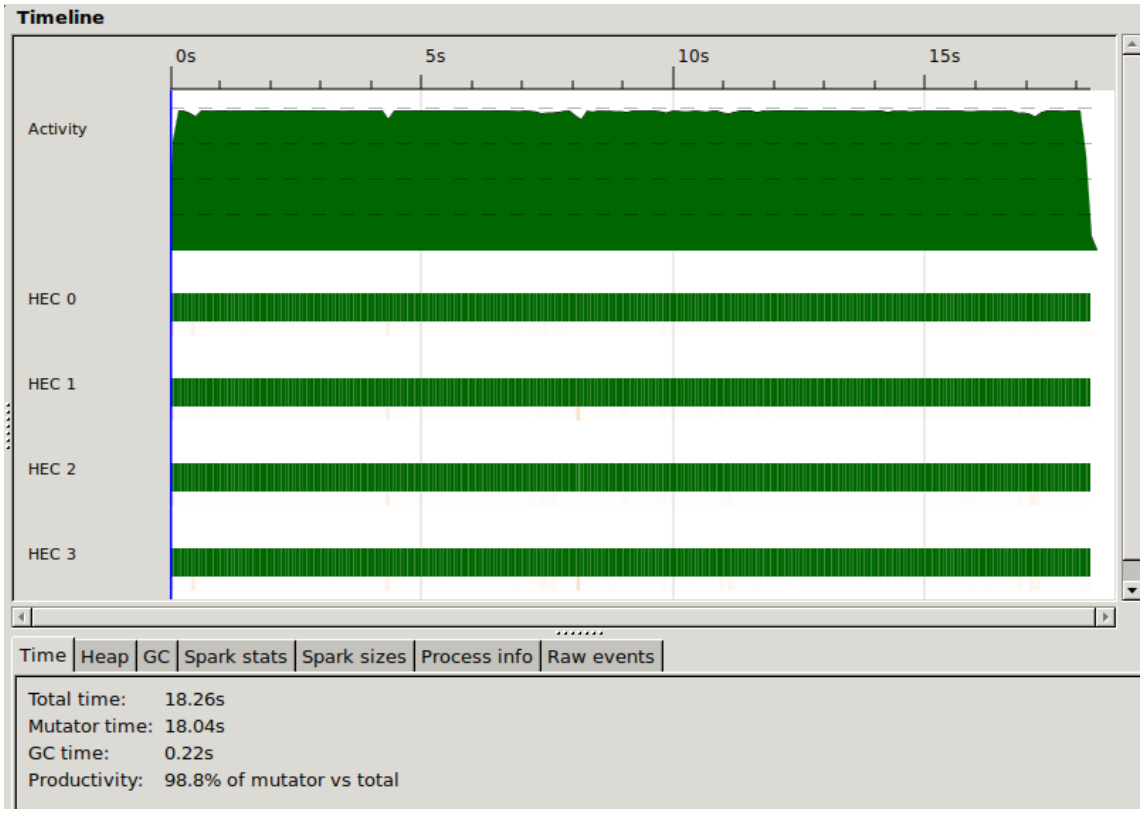


Рис. 4: Вбудовані засоби 18.26с (ThreadScope)

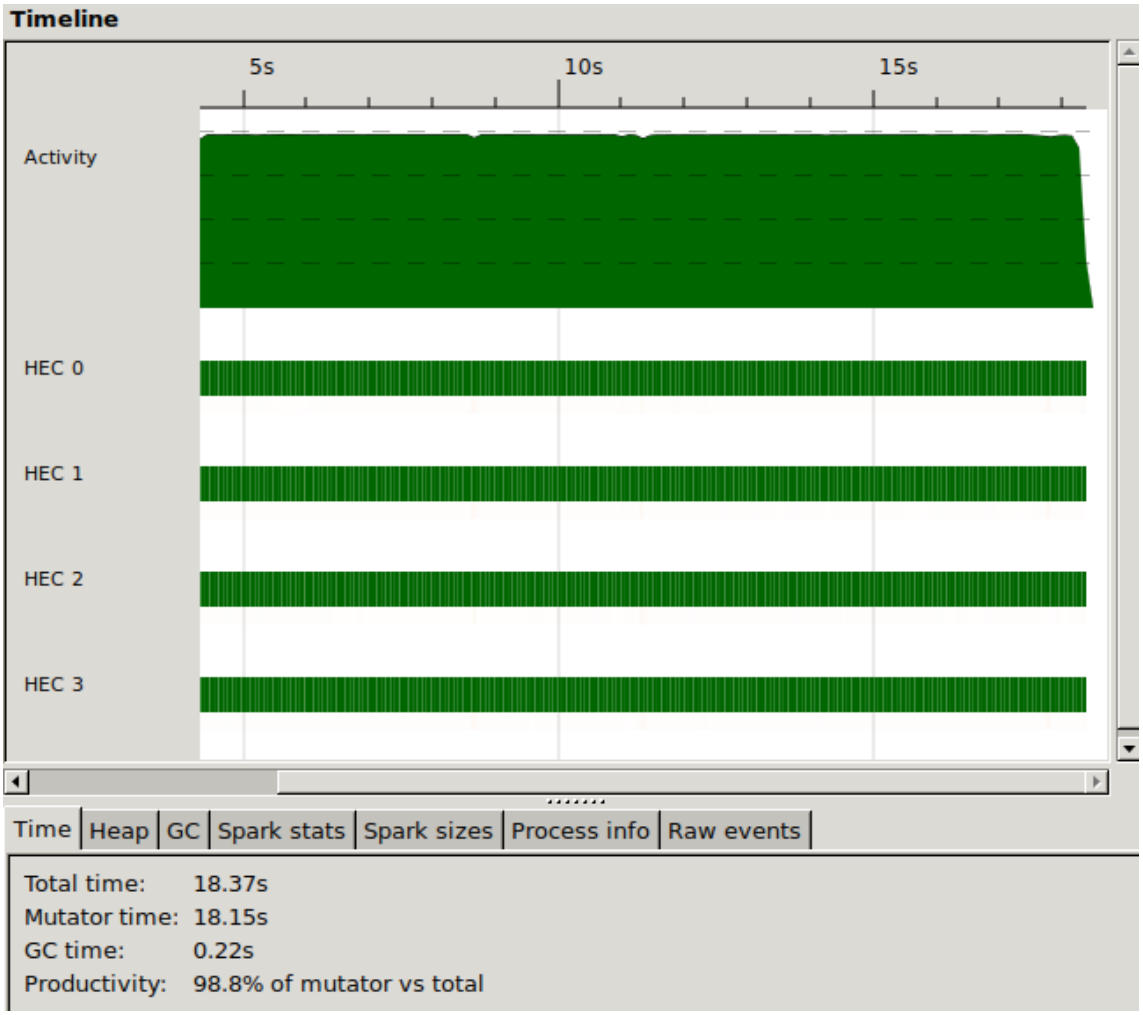


Рис. 5: Транзакційна пам'ять за 18.37с (ThreadScope)

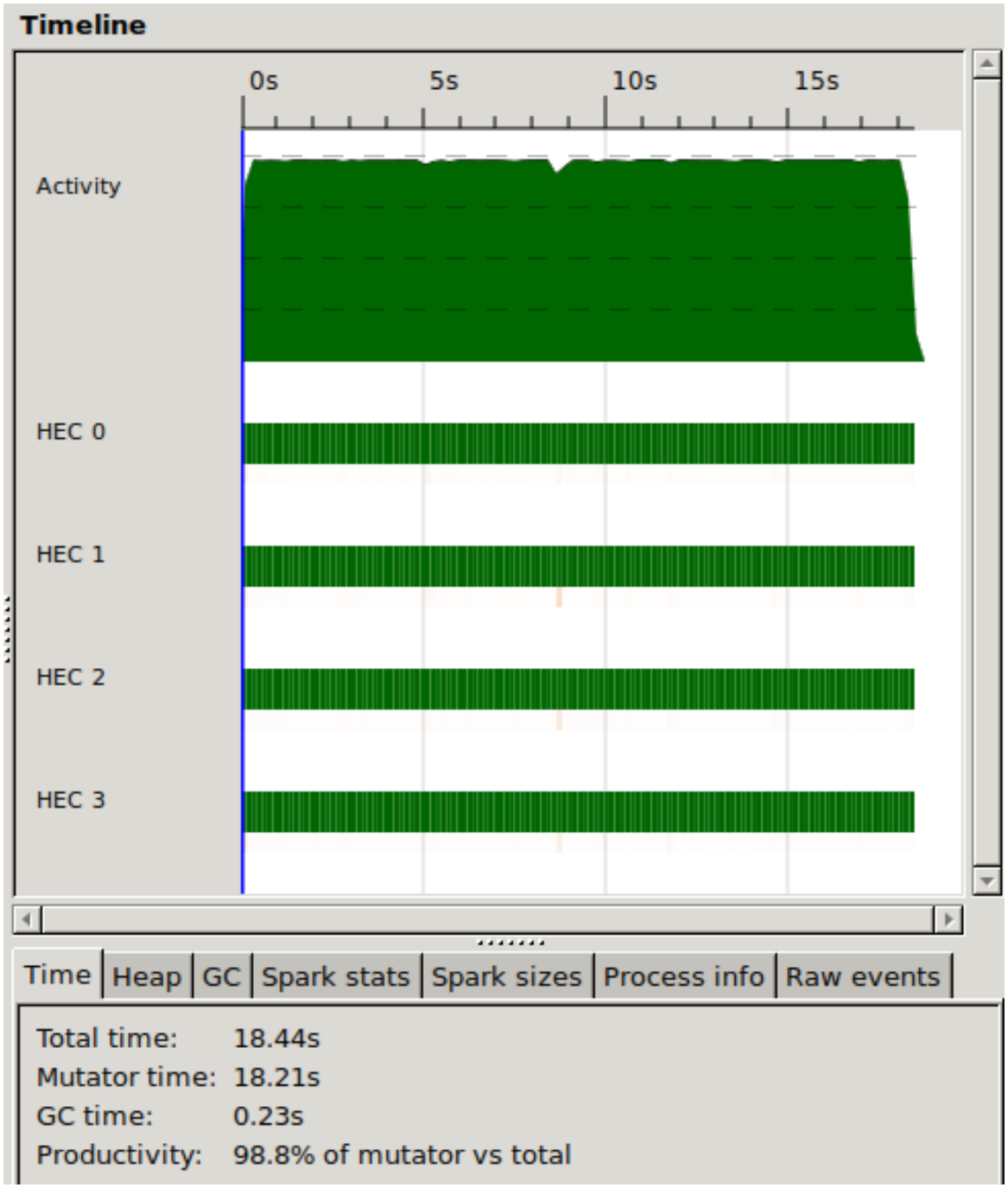


Рис. 6: Засоби Cloud Haskell за 18.44с (ThreadScope)

При невеликій взаємодії між процесами різниця незначна. Але при використанні великої кількості потоків (`nthreads` $= 10^4$) CH програє за швидкістю. З порівняння (картинка 7) можна побачити, що другий варіант (CH) потребує в

6 разів більше часу на сміттєзбірочні роботи(виклики GC позначено жовтими паличками). В той же час приріст часу для першого варіанту (STM) ледь помітно. Якщо ми заглянемо до вихідних кодів бібліотеки, то знайдемо відповідь - сама бібліотека написана з використанням тих самих вбудованих примітивів та транзакційної пам'яті. При тому, створюється набагато більше роздільних змінних (по декілька TVar-ів на актора проти лише двох для STM варіанту), збільшуючи накладні витрати.

За результатами порівняння можна відчути, що порівняно зі стандартними засобами модель акторів має ту перевагу, що, на відміну від STM та Control.Concurrent, при роботі з нею протягом всього реферату автор не отримав жодного дедлоку. Також, модель акторів є більш органічним та зрозумілим способом конкурентного програмування.

Для розподіленого програмування на Гаскелі це фактично єдиний можливий на сьогодні варіант, тож й порівнювати нема з чим[12]. Порівняння з Erlang та Scala має сенс, але відносно скромне знання автором цих мов підштовхнуло останнього до виключення цього розділу з викладу.

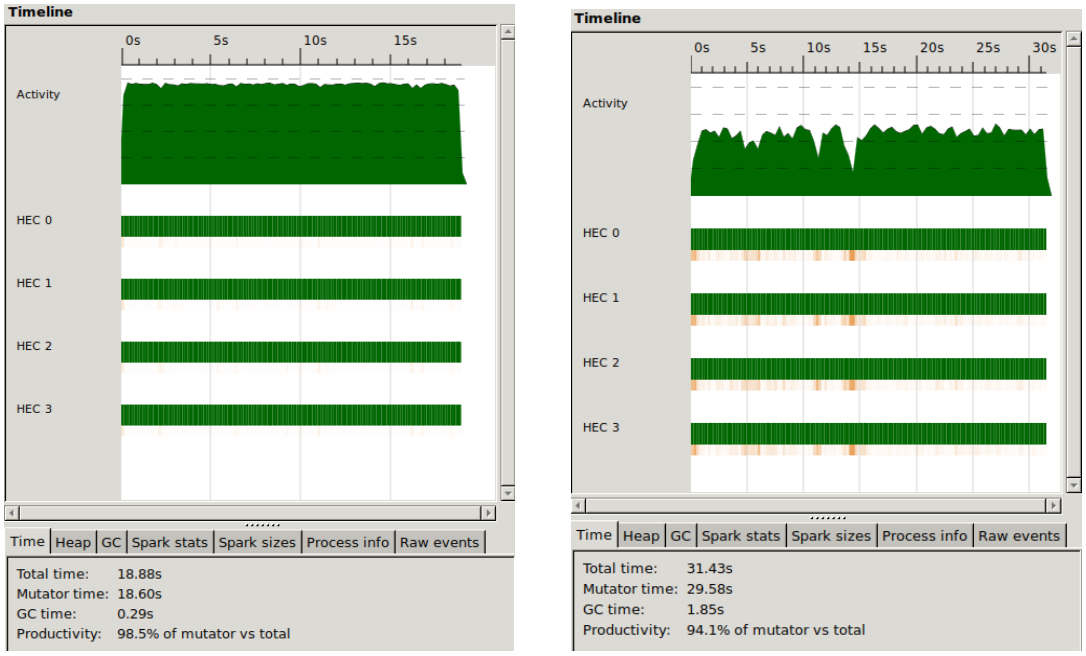
Порівняння з C, в свою чергу, не складає великої цінності, бо невтішні для гаскелю результати такого порівняння можна легко передбачити наперед.

7 Лічильник Армстронга

> One end is no end
>
> Карл Г'юїтт

Лічильник Армстронга - популярна демонстраційна розподілена програма, наведена у книзі Д. Армстронга[25]. Програма може бути або лічильником, або терміналом доступу. Кожен лічильник зберігає свій стан. З терміналу можна керувати віддаленими лічильниками - збільшити на одиницю, отримати поточне значення, або наказати лічильникові вмерти. Головна ідея цього прикладу полягає скоріше в демонстрації мережевих можливостей CH. Автор використовував в якості «кластера» свій ноутбук. Обидва вони належать до локальної мережі домашнього рутера. Ноутбук буде використовуватися в якості єдиного

[12] Дві інші спроби - Eden та GdH не розвиваються з 2014 та 2005 років відповідно. Прив'язки до MPI також покинуті автором з 2015. В свою чергу, CH активно розробляється комерційною компанією Well-Typed LLP.



(а) STM за 18.88с (б) CH за 31.43с

Рис. 7: Порівняння транзакційної пам'яті та Cloud Haskell за `nthreads` $= 10^4$

slave-а, стаціонарний ПК, конфігурацію якого детально описано на початку реферату - як master.

```
[nik@nixos:~]$ screenfetch
          ::::.    ':::::     ::::'          nik@nixos
          ':::::    ':::::.  ::::'           OS: NixOS
            :::::     ':::::.::::             Kernel: x86_64 Linux
      .......:::::..... ::::::::                    4.9.38
     ::::::::::::::::::. ::::::    ::::.      Packages: 3920
    :::::::::::::::::::: ::::::.  .:::::'     Shell: bash
            .....           ::::' :::::'      Resolution: 1366x768
          :::::             '::' :::::'       WM: XMonad
 ........:::::                ' ::::::::::.    CPU:Intel Core i5-6200U
::::::::::::                   ::::::::::::    4x 2.8GHz [30.0 C]
 :::::::::::: ..               :::::           RAM: 441MiB / 3848MiB
     .::::: .:::             :::::
    .::::::  :::::           ''''''    .....
    :::::    ':::::.  ......:::::::::::::'
     :::      :::::::. ':::::::::::::::::'
```

```
            .::::::::: ':::::::::
           .::::'':::::.     ':::::.
          .::::'   ::::.     ':::::.
         .::::      ::::      ':::::.
```

Лістинг 31: Конфігурація другого комп'ютера

Для успішного запуску необхідно по-перше вимкнути фаєрвол, а по-друге ввімкнути UDP multicast, бо саме за цим протоколом працює пошук пірів. Перше можна зробити в `/etc/nixos/configuration.nix`, задавши `networking.firewall.enable = false;` та `$ sudo nixos-rebuild switch`. Друге можна виконати як `$ sudo ifconfig lo multicast`.

Для роботи визначимо простий тип команд.

```
data Command = Increment
             | Get ProcessId
             | Suicide
             deriving (Typeable, Generic)

instance Binary Command
```

Лістинг 32: src/Armstrong.hs

Функціонал терміналу доступу. Зауважте, що на кожному терміналі 2 процеси, один для користувацького вводу, інший для виводу повідомлень.

```
master :: Backend -> Process ()
master backend = do
  pid <- spawnLocal logCounter
  run pid
  where
    usage = "Usage: I(ncrement) | G(et) | S(uicide)"

    parse i "G" = Just (Get i)
    parse _ "I" = Just Increment
    parse _ "S" = Just Suicide
    parse _  _  = Nothing

    logCounter = do
      (from, n) :: (ProcessId, Int) <- expect
      liftIO $ putStrLn $ "{" ++ show from ++ " = " ++ show n ++ "}"
```

```
      logCounter

    run pid = do
      slaves <- findSlaves backend
      liftIO $ putStrLn $ "[" ++ show (length slaves) ++ "] Enter
   message: "
      msg <- liftIO $ getLine
      case parse pid msg of
        Nothing -> do liftIO $ putStrLn usage
        Just cmd -> forM_ slaves $ flip send cmd
      run pid
```

Лістинг 33: src/Armstrong.hs

Слейви мають бути зареєстровані як slaveController, щоб пошук findSlaves їх автоматично підібрав.

```
slave :: Process ()
slave = do
  me <- getSelfPid
  register "slaveController" me
  go me (0 :: Int)
  where go me n = do
          cmd :: Command <- expect
          n' <- case cmd of
            Increment -> return (n + 1)
            Suicide -> die Suicide
            Get pid -> send pid (me, n) >> return n
          liftIO $ putStrLn $ "[" ++ show n' ++ "]"
          go me n'
```

Лістинг 34: src/Armstrong.hs

Програму можна запустити в двох режимах.

```
main :: IO ()
main = do
  args <- getArgs
  case args of
    ["master", host, port] -> do
      backend <- initializeBackend host port initRemoteTable
      node <- newLocalNode backend
```

```
      runProcess node (master backend)
    ["slave", host, port] -> do
      backend <- initializeBackend host port initRemoteTable
      node <- newLocalNode backend
      runProcess node slave
```

Лістинг 35: src/Armstrong.hs

Нижче наведено приклад користувацької сесії. Можна відчути, що актуалізація всієї мережі створює затримку приблизно в одну секунду, тож в реальних додатках доцільно використовувати більш розумні техніки виявлення падінь, наприклад, монітори.

```
[nik@nixos:~/projects/CloudRef]$ stack exec -- Armstrong master
   192.168.0.53 7000
Master started!
[1] Enter message:
A
Usage: I(ncrement) | G(et) | S(uicide)
[1] Enter message:
I
[1] Enter message:
I
[1] Enter message:
I
[1] Enter message:
G
{pid://192.168.0.90:7000:0:8 = 3}
[1] Enter message:
S
[0] Enter message:
I
[0] Enter message:
G
[1] Enter message:
I
[1] Enter message:
I
I[1] Enter message:

I[1] Enter message:
```

```
[1] Enter message:
I
[1] Enter message:
S
[0] Enter message:
```

Лістинг 36: Приклад використання - Мастер

На віддаленому комп'ютері бачимо такий результат:

```
[nik@nixos:~/projects/CloudRef]$ stack exec --
Armstrong s
[1]
[2]
[3]
[3]
Armstrong: exit-from=pid://192.168.0.90:7000:0:8

[nik@nixos:~/projects/CloudRef]$ stack exec --
Armstrong s
[1]
[2]
[3]
[4]
[5]
Armstrong: exit-from=pid://192.168.0.90:7000:0:8
```

Лістинг 37: Приклад використання - Слейв

8 Філософська вечеря

Спробуємо розв'язати класичну задачу про філософів, що обідають, з використанням Cloud Haskell. Імплементуємо тривіальне рішення з додатковим актором - столом.

Введемо синоніми типів Philosopher та Fork. Кожен філософ може або спробувати взяти пару виделок, або їх повернути.

```
type Philosopher = String
```

```
type Fork = Int

data TableInteraction = GrabForks ProcessId (Fork, Fork)
                      | ReleaseForks (Fork, Fork)
                      deriving (Show, Eq, Typeable, Generic)

instance Binary TableInteraction
```

Лістинг 38: src/Philosophers.hs

```
think :: Philosopher -> Process ()
think p = do
  say $ p ++ " is thinking"
  r <- liftIO $ randomRIO (1,100)
  liftIO $ threadDelay $ r * 100

eat :: Philosopher -> Fork -> Fork -> Process ()
eat p _ _ = do
  say $ p ++ " is eating"
  r <- liftIO $ randomRIO (1,100)
  liftIO $ threadDelay $ r * 100
```

Лістинг 39: src/Philosophers.hs

При ініціалізації, філософ очікує, що йому скажуть про його стіл та виделки. Далі в циклі філософ надсилає запит виделок столові, і очікує на них. Якщо стіл не повернув виделок, то філософ думає. Якщо виделки прийшли - він їсть та повертає виделки столові. Слід відмітити, що в гаскелі, як і в ерланзі, стандартний спосіб зберігати декотрі змінні - це рекурсія.

```
philosopher p = do
  (table, forks) <- expect
  me <- getSelfPid
  loop table forks me
  where
    loop table forks me = do
      send table $ GrabForks me forks
      maybeForks <- expect
      case maybeForks of
        Nothing -> return ()
        Just (f1, f2) -> do
          eat p f1 f2
```

```
          send table $ ReleaseForks forks
      think p
      loop table forks me
```

Лістинг 40: src/Philosophers.hs

Якщо хтось повернув виделки - додаємо їх до списку. Якщо запитав - перевіряємо їхню наявність та видаляємо при виділенні.

```
table forks = do
  interaction <- expect
  case interaction of
    ReleaseForks (f1, f2) -> table $ f1:f2:forks
    GrabForks pid fs@(f1, f2) -> do
      let (msg, forks') =
            if f1 `elem` forks && f2 `elem` forks
            then (Just fs, forks \\ [f1, f2])
            else (Nothing, forks)
      send pid msg
      table forks'
```

Лістинг 41: src/Philosophers.hs

Створюємо 7 процесів - стіл, філософи, офіціант, який проводить наших філософів[13] до столів та видає їм столові прибори.

```
main :: IO ()
main = do
  let philosophers =
        [ "Zizek"
        , "Sloterdijk"
        , "Chalmers"
        , "Dennett"
        , "MacIntyre" ]

  let forks = [1..5]
      forkSets = zip forks (tail . cycle $ forks)

  t <- liftIO createTransport
```

[13]На жаль, в класичному формулюванні філософів 5 і неможливо зберегти баланс Сили між Континентальною та Аналітичною школами

```
  node <- newLocalNode t initRemoteTable

  table <- forkProcess node $ table forks
  ps <- forM philosophers $  forkProcess node . philosopher

  waiter <- forkProcess node $ do
    forM_ (ps `zip` forkSets) $ \(p, fs) -> send p (table, fs)

  liftIO getLine
  closeTransport t
```

Лістинг 42: src/Philosophers.hs

Результати 10 секундної трапези приведені в таблиці 2.

	Їв	Думав
Жіжек	416	1087
Слотердайк	416	1129
Макінтайр	412	1099
Чалмерс	440	1135
Денет	402	1092

Табл. 2: Трапеза

9 MapReduce

MapReduce - це програмна модель та програмний каркас, що її реалізує, розроблені компанією Google для розподіленої паралельної обробки великих масивів. В дисертації Джефрі Епштейна[4] наводиться приклад MapReduce обчислення k-середніх, про який авторо стверджується, що на великих наборах даних CH обігнав Apache Hadoop. MapReduce обчислення складається з декількох маперів, які обробляють пари ключ/значення і генерують набір проміжних пар ключ/значення, та меншої кількості редукторів, які об'єднують результати обчислень з одним і тим же проміжним ключем. Цю узагальнену схему реалізує тип MapReduce.

```
data MapReduce k1 v1 k2 v2 v3 = MapReduce {
    mrMap    :: k1 -> v1 -> [(k2, v2)]
  , mrReduce :: k2 -> [v2] -> v3
```

```
  } deriving (Typeable)

localMapReduce :: forall k1 k2 v1 v2 v3. Ord k2 =>
                  MapReduce k1 v1 k2 v2 v3
               -> Map k1 v1
               -> Map k2 v3
localMapReduce mr = reducePerKey mr . groupByKey . mapPerKey mr

reducePerKey :: MapReduce k1 v1 k2 v2 v3 -> Map k2 [v2] -> Map k2 v3
reducePerKey mr = Map.mapWithKey (mrReduce mr)

groupByKey :: Ord k2 => [(k2, v2)] -> Map k2 [v2]
groupByKey = Map.fromListWith (++) . map (second return)

mapPerKey :: MapReduce k1 v1 k2 v2 v3 -> Map k1 v1 -> [(k2, v2)]
mapPerKey mr = concatMap (uncurry (mrMap mr)) . Map.toList
```

Лістинг 43: src/MapReduce/MapReduce.hs

Розподіл роботи виконується за схемою Work-stealing за допомогою черги.

```
mapperProcess :: (ProcessId, ProcessId, Closure (MapReduce String
   String String Int Int))
              -> Process ()
mapperProcess (master, workQueue, mrClosure) = do
    us <- getSelfPid
    mr <- unClosure mrClosure
    go us mr
  where
    go us mr = do
      -- Просимо роботи в черги
      send workQueue us
      say "Ask for work"

      -- Чекаємо на відповідь
      receiveWait
        [ match $ \(key, val) -> do
            say "Mapping..."
            send master (mrMap mr key val)
            say "Mapped!"
            go us mr
```

```
        , match $ \()         -> return ()
        ]

remotable ['mapperProcess]
```

Лістинг 44: src/MapReduce/MonoDistrMapReduce.hs

Головний вузол виконує 2 функції - видає роботу маперам, та проводить підсумкову редукцію.

```
distrMapReduce :: Closure (MapReduce String String String Int Int)
               -> [NodeId]
               -> Map String String
               -> Process (Map String Int)
distrMapReduce mrClosure mappers input = do
  mr     <- unClosure mrClosure
  master <- getSelfPid

  workQueue <- spawnLocal $ do
    -- Висилаємо маперам нову частину роботи
    forM_ (Map.toList input) $ \(key, val) -> do
      them <- expect
      send them (key, val)

    -- Завершимо всі mappeги-
    replicateM_ (length mappers) $ do
      them <- expect
      send them ()

  -- Запускаємо мапери
  say "Waking up mappers..."
  forM_ mappers $ \nid -> do
    spawn nid ($(mkClosure 'mapperProcess) (master, workQueue,
   mrClosure))

  partials <- replicateM (Map.size input) expect

  say "Reducing..."
  -- Маємо єдиний редуктор на цій ноді
  return (reducePerKey mr . groupByKey . concat $ partials)
```

Лістинг 45: src/MapReduce/MonoDistrMapReduce.hs

9.1 Підрахунок частот

Будемо використовувати MapReduce для підрахунку частот слів в файлі.

```
type Document  = String
type Word      = String
type Frequency = Int

countWords :: MapReduce FilePath Document Word Frequency Frequency
countWords = MapReduce {
    mrMap    = const (map (, 1) . words)
  , mrReduce = const sum
  }

localCountWords :: Map FilePath Document -> Map Word Frequency
localCountWords = localMapReduce countWords

countWords_ :: () -> MapReduce FilePath Document Word Frequency
   Frequency
countWords_ () = countWords

remotable ['countWords_]

distrCountWords :: [NodeId] -> Map FilePath Document -> Process (Map
   Word Frequency)
distrCountWords = distrMapReduce ($(mkClosure 'countWords_) ())
```

Лістинг 46: src/MapReduce/CountWords.hs

Головна програма реалізує простий користувацький інтерфейс. Використовуються стандартні мастер та слейв функції.

```
main :: IO ()
main = do
  args <- getArgs

  case args of
    -- Локальне рахування слів
    "local" : "count" : files -> do
      input <- constructInput files
      print $ CountWords.localCountWords input
```

```
    -- Розподілений підрахунок слів
    "master" : host : port : "count" : files -> do
      input   <- constructInput files
      backend <- initializeBackend host port rtable
      startMaster backend $ \slaves -> do
        result <- CountWords.distrCountWords slaves input
        liftIO $ print result

    -- slave
    "slave" : host : port : [] -> do
      backend <- initializeBackend host port rtable
      startSlave backend
```

Лістинг 47: src/MapReduce/SimpleLocalnet.hs

На жаль, як можна побачити з приведених нижче результатів, експеримент виявився невдалим. Задачу підрахування частот для 7 файлів розміром 4.5 мегабайт пришвидчити не вдалося. Можливі причини невдачі - великі накладні витрати на використання мережі на редукцію, помилка при проектуванні програми, відносно легкі процеси віддані маперам.

```
time stack exec -- MapReduce master 192.168.0.53 60000 count a b c d e f g
    >> /dev/null
Thu Nov 23 15:40:21 UTC 2017 pid://192.168.0.53:60000:0:8: Begin
Thu Nov 23 15:40:21 UTC 2017 pid://192.168.0.53:60000:0:8: Waking up
    mappers...
Thu Nov 23 15:40:21 UTC 2017 pid://192.168.0.90:5001:0:12: Ask for work
Thu Nov 23 15:40:21 UTC 2017 pid://192.168.0.90:5002:0:12: Ask for work
Thu Nov 23 15:40:21 UTC 2017 pid://192.168.0.90:5003:0:12: Ask for work
Thu Nov 23 15:40:21 UTC 2017 pid://192.168.0.90:5004:0:12: Ask for work
Thu Nov 23 15:40:21 UTC 2017 pid://192.168.0.53:60003:0:15: Ask for work
Thu Nov 23 15:40:21 UTC 2017 pid://192.168.0.53:60006:0:15: Ask for work
Thu Nov 23 15:40:21 UTC 2017 pid://192.168.0.53:60005:0:15: Ask for work
Thu Nov 23 15:40:21 UTC 2017 pid://192.168.0.53:60004:0:15: Ask for work
Thu Nov 23 15:40:23 UTC 2017 pid://192.168.0.90:5001:0:12: Mapping...
Thu Nov 23 15:40:25 UTC 2017 pid://192.168.0.90:5002:0:12: Mapping...
Thu Nov 23 15:40:26 UTC 2017 pid://192.168.0.90:5003:0:12: Mapping...
Thu Nov 23 15:40:27 UTC 2017 pid://192.168.0.90:5004:0:12: Mapping...
Thu Nov 23 15:40:28 UTC 2017 pid://192.168.0.53:60003:0:15: Mapping...
Thu Nov 23 15:40:28 UTC 2017 pid://192.168.0.53:60006:0:15: Mapping...
Thu Nov 23 15:40:28 UTC 2017 pid://192.168.0.90:5001:0:12: Mapped!
Thu Nov 23 15:40:28 UTC 2017 pid://192.168.0.90:5001:0:12: Ask for work
Thu Nov 23 15:40:28 UTC 2017 pid://192.168.0.53:60005:0:15: Mapping...
Thu Nov 23 15:40:29 UTC 2017 pid://192.168.0.53:60003:0:15: Mapped!
```

```
Thu Nov 23 15:40:29 UTC 2017 pid://192.168.0.53:60003:0:15: Ask for work
Thu Nov 23 15:40:30 UTC 2017 pid://192.168.0.53:60006:0:15: Mapped!
Thu Nov 23 15:40:30 UTC 2017 pid://192.168.0.53:60006:0:15: Ask for work
Thu Nov 23 15:40:30 UTC 2017 pid://192.168.0.53:60005:0:15: Mapped!
Thu Nov 23 15:40:30 UTC 2017 pid://192.168.0.53:60005:0:15: Ask for work
Thu Nov 23 15:40:31 UTC 2017 pid://192.168.0.90:5002:0:12: Mapped!
Thu Nov 23 15:40:31 UTC 2017 pid://192.168.0.90:5002:0:12: Ask for work
Thu Nov 23 15:40:32 UTC 2017 pid://192.168.0.90:5003:0:12: Mapped!
Thu Nov 23 15:40:32 UTC 2017 pid://192.168.0.90:5003:0:12: Ask for work
Thu Nov 23 15:40:32 UTC 2017 pid://192.168.0.90:5004:0:12: Mapped!
Thu Nov 23 15:40:32 UTC 2017 pid://192.168.0.90:5004:0:12: Ask for work
Thu Nov 23 15:40:36 UTC 2017 pid://192.168.0.53:60000:0:8: Reducing...

real    0m43.772s
user    0m36.316s
sys     0m1.887s
```

Лістинг 48: 4 локальних та 4 віддалених мапера

```
time stack exec -- MapReduce master 192.168.0.53 60000 count a b c d e f g
   >> /dev/null
Thu Nov 23 15:44:21 UTC 2017 pid://192.168.0.53:60000:0:8: Begin
Thu Nov 23 15:44:21 UTC 2017 pid://192.168.0.53:60000:0:8: Waking up
   mappers...
Thu Nov 23 15:44:21 UTC 2017 pid://192.168.0.53:60003:0:16: Ask for work
Thu Nov 23 15:44:21 UTC 2017 pid://192.168.0.53:60005:0:16: Ask for work
Thu Nov 23 15:44:21 UTC 2017 pid://192.168.0.53:60004:0:16: Ask for work
Thu Nov 23 15:44:21 UTC 2017 pid://192.168.0.53:60006:0:16: Ask for work
Thu Nov 23 15:44:23 UTC 2017 pid://192.168.0.53:60003:0:16: Mapping...
Thu Nov 23 15:44:24 UTC 2017 pid://192.168.0.53:60005:0:16: Mapping...
Thu Nov 23 15:44:24 UTC 2017 pid://192.168.0.53:60004:0:16: Mapping...
Thu Nov 23 15:44:25 UTC 2017 pid://192.168.0.53:60006:0:16: Mapping...
Thu Nov 23 15:44:25 UTC 2017 pid://192.168.0.53:60003:0:16: Mapped!
Thu Nov 23 15:44:25 UTC 2017 pid://192.168.0.53:60003:0:16: Ask for work
Thu Nov 23 15:44:26 UTC 2017 pid://192.168.0.53:60005:0:16: Mapped!
Thu Nov 23 15:44:26 UTC 2017 pid://192.168.0.53:60005:0:16: Ask for work
Thu Nov 23 15:44:26 UTC 2017 pid://192.168.0.53:60004:0:16: Mapped!
Thu Nov 23 15:44:26 UTC 2017 pid://192.168.0.53:60004:0:16: Ask for work
Thu Nov 23 15:44:27 UTC 2017 pid://192.168.0.53:60006:0:16: Mapped!
Thu Nov 23 15:44:27 UTC 2017 pid://192.168.0.53:60006:0:16: Ask for work
Thu Nov 23 15:44:28 UTC 2017 pid://192.168.0.53:60003:0:16: Mapping...
Thu Nov 23 15:44:30 UTC 2017 pid://192.168.0.53:60003:0:16: Mapped!
Thu Nov 23 15:44:30 UTC 2017 pid://192.168.0.53:60003:0:16: Ask for work
Thu Nov 23 15:44:29 UTC 2017 pid://192.168.0.53:60005:0:16: Mapping...
```

```
Thu Nov 23 15:44:30 UTC 2017 pid://192.168.0.53:60005:0:16: Mapped!
Thu Nov 23 15:44:30 UTC 2017 pid://192.168.0.53:60005:0:16: Ask for work
Thu Nov 23 15:44:29 UTC 2017 pid://192.168.0.53:60004:0:16: Mapping...
Thu Nov 23 15:44:31 UTC 2017 pid://192.168.0.53:60004:0:16: Mapped!
Thu Nov 23 15:44:31 UTC 2017 pid://192.168.0.53:60004:0:16: Ask for work
Thu Nov 23 15:44:35 UTC 2017 pid://192.168.0.53:60000:0:8: Reducing...

real    0m40.234s
user    0m35.479s
sys     0m1.929s
```

Лістинг 49: 4 локальних мапери

```
time stack exec -- MapReduce local count a b c d d e f g >> /dev/null

real    0m34.580s
user    0m33.182s
sys     0m1.448s
```

Лістинг 50: Послідовний варіант

9.2 Факторизація

Невдача в першому досліді підштовхнула автора до нового експерименту, в якому частина Map буде більш вагомою порівняно з reduce. Нехай це буде факторизація чисел з одного з перших прикладів. В попередньому прикладі для простоти використовувалась мономорфна функція distrMapReduce. Зробимо її поліморфною.

```
matchDict :: forall a b. SerializableDict a -> (a -> Process b) ->
   Match b
matchDict SerializableDict = match

sendDict :: forall a. SerializableDict a -> ProcessId -> a ->
   Process ()
sendDict SerializableDict = send

sdictProcessIdPair :: SerializableDict (ProcessId, ProcessId)
sdictProcessIdPair = SerializableDict

mapperProcess :: forall k1 v1 k2 v2 v3.
                 SerializableDict (k1, v1)
```

```
              -> SerializableDict [(k2, v2)]
              -> (ProcessId, ProcessId)
              -> MapReduce k1 v1 k2 v2 v3
              -> Process ()
mapperProcess dictIn dictOut (master, workQueue) mr = getSelfPid >>=
   go
  where
    go us = do
      say "Asking for work..."
      send workQueue us

      receiveWait
        [ matchDict dictIn $ \(key, val) -> do
            say "Mapping..."
            sendDict dictOut master (mrMap mr key val)
            say "Mapped!"
            go us
        , match $ \() -> do
            say "Died"
            return ()
        ]

remotable ['mapperProcess, 'sdictProcessIdPair]

mapperProcessClosure :: forall k1 v1 k2 v2 v3.
                        (Typeable k1, Typeable v1, Typeable k2,
   Typeable v2, Typeable v3)
                     => Static (SerializableDict (k1, v1))
                     -> Static (SerializableDict [(k2, v2)])
                     -> Closure (MapReduce k1 v1 k2 v2 v3)
                     -> ProcessId
                     -> ProcessId
                     -> Closure (Process ())
mapperProcessClosure dictIn dictOut mr master workQueue =
    closure decoder (encode (master, workQueue)) `closureApply` mr
  where
    decoder :: Static (ByteString -> MapReduce k1 v1 k2 v2 v3 ->
   Process ())
    decoder =
        ($(mkStatic 'mapperProcess) `staticApply` dictIn
```

```
   `staticApply` dictOut)
     `staticCompose`
       staticDecode $(mkStatic 'sdictProcessIdPair)

distrMapReduce :: forall k1 k2 v1 v2 v3 a.
                  (Serializable k1, Serializable v1,
                   Serializable k2, Serializable v2,
                   Serializable v3, Ord k2)
               => Static (SerializableDict (k1, v1))
               -> Static (SerializableDict [(k2, v2)])
               -> Closure (MapReduce k1 v1 k2 v2 v3)
               -> [NodeId]
               -> ((Map k1 v1 -> Process (Map k2 v3)) -> Process a)
               -> Process a
distrMapReduce dictIn dictOut mr mappers p = do
  mr'    <- unClosure mr
  master <- getSelfPid

  workQueue <- spawnChannelLocal $ \queue -> do
    let go :: Process ()
        go = do
          mWork <- receiveChan queue
          case mWork of
            Just (key, val) -> do
              them <- expect
              send them (key, val)
              go
            Nothing -> do
              replicateM_ (length mappers) $ do
                them <- expect
                send them ()

              send master ()
    go

  say "Waking up mappers..."
  let workQueuePid = sendPortProcessId (sendPortId workQueue)
  forM_ mappers $ \nid ->
    spawn nid (mapperProcessClosure dictIn dictOut mr master
   workQueuePid)
```

```
  let iteration :: Map k1 v1 -> Process (Map k2 v3)
      iteration input = do
        mapM_ (sendChan workQueue . Just) (Map.toList input)

        partials <- replicateM (Map.size input) expect

        say "Reducing..."
        return (reducePerKey mr' . groupByKey . concat $ partials)

  result <- p iteration

  sendChan workQueue Nothing
  expect :: Process ()

  return result
```

Лістинг 51: src/MapReduce/PolyDistrMapReduce.hs

```
factorize :: Integer -> Integer -> [Integer]
factorize _ 1 = []
factorize d n
    | d * d > n = [n]
    | n `mod` d == 0 = d : factorize d (n `div` d)
    | otherwise = factorize (d + 1) n

primeFactors :: Integer -> [Integer]
primeFactors = factorize 2

countFactors :: MapReduce Integer Integer () Integer Integer
countFactors = MapReduce {
    mrMap = \k v ->
        map (\n -> ((), toInteger $ length $ primeFactors n))
    [k..v-1]
  , mrReduce = const sum
  }


localCountFactors = localMapReduce countFactors

countFactors_ :: () -> MapReduce Integer Integer () Integer Integer
```

```
countFactors_ () = countFactors

dictIn :: SerializableDict (Integer, Integer)
dictIn = SerializableDict

dictOut :: SerializableDict [((), Integer)]
dictOut = SerializableDict

remotable ['countFactors_, 'dictIn, 'dictOut]

distrCountFactors :: [NodeId] -> Map Integer Integer -> Process (Map
   () Integer)
distrCountFactors nodes m =
  distrMapReduce
    $(mkStatic 'dictIn)
    $(mkStatic 'dictOut)
    ($(mkClosure 'countFactors_) ())
    nodes
    (\iteration -> iteration m)
```

Лістинг 52: src/MapReduce/CountFactors.hs

```
    ...
    "local" : "factors" : from : to : part : _ -> do
      let input = constructInput2 from to part
      print $ CountFactors.localCountFactors input

    "master" : host : port : "factors"
      : from : to : part : _ -> do
      let input = constructInput2 from to part
      backend <- initializeBackend host port rtable
      startMaster backend $ \slaves -> do
        say $ "Begin [" ++ show (length slaves) ++ "]"
        result <- CountFactors.distrCountFactors slaves input
        liftIO $ print $ result

...

constructInput2 from to part =
  let (a,b,p) = (read from, read to, read part)
      xs = [a, a+p..b]
```

```
  in Map.fromList (zip xs (tail xs))
```

Лістинг 53: src/MapReduce/SimpleLocalnet.hs

Округлені результати наведено в табл.3.

Обчислювання	Від	До	Частина	Нод на к1	Нод на к2	Час
Локальне	1M	3M	100K	0	0	1 хв 40 с
Розподілене	1M	3M	100K	4	0	35 с
Розподілене	1M	3M	100K	0	4	36 с
Розподілене	1M	3M	100K	4	4	24 с
Розподілене	1M	3M	500K	4	0	39 с

Табл. 3: Підрахунок кількості простих множників

10 Розподілене KV-сховище

Даний приклад націлено не на демонстрацію пришвидшення паралельних обчислень, а на ілюстрацію того, як можна зробити деякі важливі обчислення відказостійкими. Нехай є декілька комп'ютерів (для простоти промоделюємо їх як окремі процеси). Ми хочемо зберігати на кожному декотру інформацію, а також мати можливість її зчитувати. Для простоти будемо зберігати її лише в оперативній пам'яті. Хоча тут маємо цілком шкільний приклад, розподілені сховища даних є одними з найважливіших застосувань акторних обчислень.

Створимо необхідні типи.

```
type Key   = String
type Value = String

data Request = GET Key (SendPort (Maybe Value))
             | SET Key Value
             deriving (Typeable,Generic)

instance Binary Request
```

Лістинг 54: src/KV/Request.hs

Функція, яку буде запущено на віддалених нодах. SET запроси обробляються напряму, GET - з використанням каналу.

```
worker :: Process ()
worker = loop Map.empty
    where
        loop db = do
            r <- expect :: Process Request
            case r of
              SET k v -> do
                let newdb = Map.insert k v db
                loop newdb
              GET k s -> do
                let mval = Map.lookup k db
                sendChan s mval
                loop db

remotable ['worker]
```

Лістинг 55: src/KV/Worker.hs

Функції для мережевої взаємодії:

```
set :: Database -> Key -> Value -> Process ()
set db k v = send db (SET k v)

get :: Database -> Key -> Process (Maybe Value)
get db k = do
    (s,r) <- newChan
    let req = GET k s
    send db (GET k s)
    receiveChan r
```

Лістинг 56: src/KV/Database.hs

Мастер процес, тобто база даних, працює так. Спочатку від'єднується паралельний процес, який моніторить кількість живих рабів (доки є хоча б один живий), а якщо хтось від'єднався - сповіщає користувача. Зауважимо, що для роботи треба мінімум 2 вузли. Живі раби зберігаються в транзакційній змінній.

```
database :: [NodeId] -> Process ()
database nodes = do
    ps <- mapM (\n -> spawn n $(mkStaticClosure 'worker)) nodes

    tps <- liftIO $ newTVarIO (pairs ps)
```

```
    spawnLocal $ do
      mapM_ monitor ps
      whileM_  (do
                  pids <- liftIO $ atomically $ readTVar tps
                  return $ length pids > 0
               )$
        receiveWait
        [ match $ \(ProcessMonitorNotification ref deadpid reason)
   -> do
                   liftIO $ atomically $ modifyTVar' tps $ map
   (delete deadpid)
                   say $ printf "process %s died: %s" (show
   deadpid) (show reason)
        ]

    forever $ do
      r <- expect :: Process Request
      ps' <- liftIO $ atomically $ readTVar tps
      case r of
        GET k s-> mapM_ (\p -> send p r) (npid k ps')
        SET k v -> mapM_ (\p -> send p r) (npid k ps')
    where
        pairs [] = []
        pairs [x] = []
        pairs (x:y:xs) = [x,y] : pairs xs
        npid k ps = ps !! (ord (head k) `mod` length ps)
```

Лістинг 57: src/KV/Database.hs

Проста точка входу дозволяє вводити команди. Функція distribMain обробляє аргументи безпосередньо програми за допомогою нудного boilerplate коду, який немає сенсу приводити. Для спрощення використовуються стандартні функції слейвів та мастерів.

```
main = distribMain master rcdata

master :: [NodeId] -> Process ()
master peers = do
  db <- createDB peers
  liftIO $ print peers
  set db "a" "1"
```

```
  forever $ do
    liftIO $ putStrLn "ENTER COMMAND:"
    cmd <- liftIO getLine
    case words cmd of
      ["SET", k, v] -> do
        set db k v
      ["GET", k] -> do
        v <- get db k
        liftIO $ putStrLn $ "response: " ++ show v
      _ -> liftIO $ putStrLn $ "usage: SET k v | GET k"

  return ()
```

Лістинг 58: src/KV/Main.hs

Для демонстрації запустимо декілька слейвів `stack exec -- KV slave 44447 &` та запустимо мастер-термінал. Потім спробуємо зупинити декілька процесів, і перевірити працездатність та цілісність сховища. Можна також погратися з декількома точками доступу (Універсальні слейви не будуть змішувати дані різних мастерів).

```
$ stack exec -- KV master
[nid://127.0.0.1:44447:0,nid://127.0.0.1:44446:0,
nid://127.0.0.1:44427:0,nid://127.0.0.1:44448:0]
ENTER COMMAND:
GET a
response: Just "1"
ENTER COMMAND:
GET b
response: Nothing
ENTER COMMAND:
SET a 12
ENTER COMMAND:
GET a
response: Just "12"
ENTER COMMAND:
Wed Nov 22 19:56:37 UTC 2017 pid://127.0.0.1:44444:0:10: process
    pid://127.0.0.1:44446:0:10 died: DiedDisconnect
GET a
response: Just "12"
ENTER COMMAND:
```

```
SET a 5
ENTER COMMAND:
GET a
response: Just "5"
ENTER COMMAND:
```

Лістинг 59: Приклад сесії

11 Допоміжна довідка

Лямбда числення

Більш розгорнуте викладення можна знайти, наприклад у[26].

Синтаксис лямбда числення задається наступною граматикою:

```
Нескінченна
 перелічувана множина змінних:
V ::= x | y | x1 | ...Термом

 є змінна, абстракція, або аплікація:
M, N ::= V
     | (\V -> M)
     | (M N)
```

Лістинг 60: Синтаксис простого безтипового лямбда числення

При цьому, абстракція є правоасоціативною, аплікація - лівоасоциативною.

Множина вільних змінних `FV(M)` терму `N` задається рекурентно:

```
FV(x) = {x}
FV(\x -> P) = FV(M) \ {x}
FV(P Q) = FV(P) ∪ FV (Q)
```

Лістинг 61: Вільні змінні

Вільна змінна не зв'язана жодною лямбда абстракцією, наприклад в термі `(\f -> (\x -> f x))` всі змінні зв'язані, але з точки зору його внутрішнього терму `(\x -> f x)` змінна y є вільною.

Функція (тобто λ-терм) є замкненою, якщо множина її вільних змінних є пустою.

Підстановкою терма замість змінної в термі визначають так:

```
x[x := N] = N
y[x := N] = y
(P Q)[x := N] = P[x := N] Q[x := N]
(\x -> P)[x := N] = \x -> P
(\y -> P)[x := N] = \y -> P[x := N]
```

Лістинг 62: Вільні змінні

Підстановка вважається правильною, якщо жодна вільна змінна не стає зв'язаною, зв'язана - вільною.

Відношенням бета-редукції => називають таке найменше відношення на множині λ-термів, що

- `(\x -> P) Q => P[x:=Q]`
- Якщо `P => P'`, то `\x -> P => \x -> P'`
- Якщо `P => P'`, то `P Q => P' Q`
- Якщо `P => P'`, то `Q P => Q P'`

Ліву частину редукції називають редексом, праву - згортком. Якщо терм не є редексом, то кажуть, що він знаходиться у нормальній формі.

Багатошаговою бета-редукцією =>> називають транзитивно-рефлексивне замикання =>

Бета-еквівалентністю називають транзитивно-рефлексивно-симетричне замикання =>

System FC

Синтаксис Гаскеля зводиться до синтаксису System FC[27] - формальної системи поліморфного лямбда числення, яка розширює System F свідками приведення (Coercion).

Нижче приведено визначення виразу мови GHC Core, взяте з репозиторію вихідого коду GHC[28].

```
data Expr b
  = Var   Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Tick  (Tickish Id) (Expr b)
  | Type  Type
  | Coercion Coercion
  deriving Data
```

Лістинг 63: CoreSyn.hs

Замикання

Замиканням відношення R відносно властивості P називають таке мінімальне R^*, що $R \subset R^*$, та для будь якої пари $(x, y) \in R^*$ виконується $P(x, y)$. Під замиканням функції у функціональному програмуванні розуміють саму цю функцію разом з її оточенням, яке складається з її вільних змінних. Наприклад, замиканням функції `\x -> (x z) (y x)` буде пара `<(\x -> (x z) (y x)), [y; z]>`.

Лінива семантика

Call-by-need семантика означає, що редуціюються лише ті терми, які потрібно, а не всі, на які вказав програміст. Продемонструємо це у GHCi

```
λ> :set +s
λ> fac n = product [1..n]
(0.00 secs, 0 bytes)
λ> xs = replicate 10000 (fac 25000)
(0.00 secs, 0 bytes)
λ> length xs
10000
(0.01 secs, 1,052,640 bytes)
λ> fac 25000 `mod` 2
0
(1.06 secs, 597,515,792 bytes)
```

```
λ> sum xs `mod` 2
0
(1.93 secs, 1,010,760,824 bytes)
```

Лістинг 64: Ліниві обчислення

Таким чином, факторіал 25000! не вираховується тисячу разів, суттєво вкорочуючи час виконання програми. Зауважте також, що всі операції виконуються над необмеженим типом Integral, втрати даних через переповнення не виходить (на відміну від типу Int).

Втім, гаскель має спеціальні засоби для вказання на строгий порядок редукцій.

Переписування графів

Гаскель є системою лінивого переписування графів. Це означає, що кожну програму можна представити у вигляді графу, вузлами якого є санки(редекси), висячими вершинами - значення в нормальній формі. На кожному шазі виконання програми цей граф зменьшується (редуціюється). Пи цьому, синтаксично однакові вирази відображаються в єдиний вузол графу. Приклад редукції наведено у Рис. 8 на сторінці 61.

12 Висновок

За результатом роботи автор може сформувати відносно суб'єктивний список приємностей та неприємностей, які він примітив в розглянутої технології.

Приємності:

- Всі переваги Гаскеля як мови, які довго перелічувати
- Невеликі та порівняно зрозумілі[14] вихідні коди
- Чудове теоретичне підгрунтя
- Документація
- Використовується в продакшн[15] (iohk.io та ін.)

[14]Звісно що з Erlang OTP та MPI

[15]Це велика рідкість для Haskell бібліотек

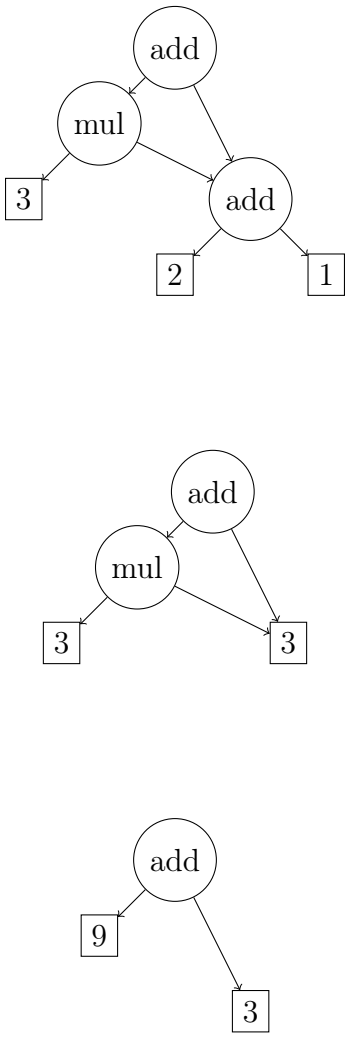


Рис. 8: Редукція терму `(add (mul 3 (add 1 2)) (add 1 2))`

Неприємності:

- Швидкодія Гаскелю
- Мала популярність
- Невелика кількість навчальних матеріалів

Перелік посилань

[1] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, 2013. ISBN: 0471935670.

[2] Саймон Марлоу (пер. В. Брагилевский). *Параллельное и конкурентное программирование на языке Haskell*. ДМК Пресс, 2014. ISBN: 978-5-94074-984-4.

[3] Simon Peyton-Jones Jeff Epstein Andrew P. Black. *Towards Haskell in the Cloud*. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/remote.pdf.

[4] Jeffrey Epstein. *MPhil thesis, Functional programming for the data centre*. 2011. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/epstein-thesis.pdf.

[5] Joe Armstrong. *PhD thesis, Making reliable distributed systems in the presence of software errors*. 2011. URL: http://erlang.org/download/armstrong_thesis_2003.pdf.

[6] *Getting NixOS*. URL: https://nixos.org/nixos/download.html.

[7] *Raspberry Pi in a Haskell Cloud*. URL: http://web.archive.org/web/20170113133119/http://alenribic.com/posts/2012-08-17-raspberry-pi-in-a-haskell-cloud.html.

[8] *Using Cloud Haskell in HPC Cluster*. URL: http://malcodigo.blogspot.com.es/2012/10/using-cloud-haskell-in-hpc-cluster.html.

[9] Микита Юрченко. *Мої конфігураційні файли*. URL: https://github.com/zelinskiy/dotfiles/tree/a07000319a7782a1477e7b68291d859b67c577b9.

[10] Микита Юрченко. *Вихідні коди прикладів*. URL: https://github.com/zelinskiy/CloudHaskell.

[11] *haskell-mode manual*. URL: http://haskell.github.io/haskell-mode/manual/latest/.

[12] *Get Emacs*. URL: https://www.gnu.org/software/emacs/.

[13] *Installing haskell-mode*. URL: https://github.com/haskell/haskell-mode/blob/9018ad5cac0b1b1b5e0a51586027fb0ca4076b1a/README.md.

[14] *GHC User's Guide*. URL: https://downloads.haskell.org/~ghc/6.6/docs/html/users_guide/ghci-commands.html.

[15] Gabriel Gonzalez. *Scrap your type classes*. 2012. URL: http://www.haskellforall.com/2012/05/scrap-your-type-classes.html.

[16] Simon Peyton Jones Paul Hudak John Hughes. *A History of Haskell: Being Lazy With Class*. 2007. URL: http://haskell.cs.yale.edu/wp-content/uploads/2011/02/history.pdf.

[17] Жиль Делёз. *Складка. Лейбниц и барокко*. Логос, 1998.

[18] Simon Peyton-Jones. *Towards haskell in the cloud, talk at Scala Days 2012*. URL: https://skillsmatter.com/skillscasts/3241-haskell-cloud.

[19] Well-Typed Ltd. *Cloud Haskell Semantics*. URL: https://haskell-distributed.github.io/static/semantics.pdf.

[20] Hans Svensson et al. *A Unified Semantics for Future Erlang*. 2010. URL: http://happy-testing.com/hans/papers/EW2010-UnifiedSemantics.pdf.

[21] Carl Hewitt. *Actor Model of Computation: Scalable Robust Information Systems*. 2015. URL: https://arxiv.org/abs/1008.1459.

[22] *Hewitt, Meijer and Szyperski: The Actor Model (everything you wanted to know...)* URL: https://www.youtube.com/watch?v=7erJ1DV_Tlo.

[23] Henry Baker Carl Hewitt. *Laws for Communicating Parallel Processes*. 1977. URL: https://dspace.mit.edu/handle/1721.1/41962.

[24] Akinori Yonezawa. *Doctoral thesis, Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics*. 1977. URL: http://www.dtic.mil/docs/citations/ADA051149.

[25] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, 2007. ISBN: 978-1-93435-600-5.

[26] Д. Н. Москвин. *Лямбда исчисление*. URL: https://compscicenter.ru/media/slides/func_prog_2014_spring/2014_02_14_func_prog_2014_spring_2.pdf.

[27] Richard Eisenberg. *System FC, as implemented in GHC*. URL: https://github.com/ghc/ghc/raw/master/docs/core-spec/core-spec.pdf.

[28] *GHC Source Mirror - GHC Core*. URL: https://github.com/ghc/ghc/blob/f63bc730c7ea42ca6882f8078eb86be8bf1cc6ad/compiler/coreSyn/CoreSyn.hs.