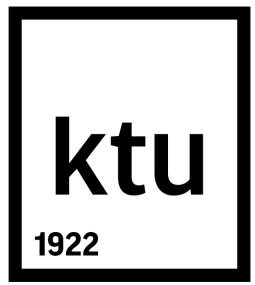
Funkcinio ir lygiagrečiojo programavimo sąsaja

Karolis Ryselis

Kauno Technologijos Universitetas



You have shot yourself in the foot using Haskell, but nothing happens unless you start walking.

Paskaitos turinys

ktu 1922

Funkcinio programavimo kalbų savybės

- Funkcinio programavimo kalbos pasižymi tokiomis savybėmis:
  - Nėra kintamųjų ir keičiamos būsenos;
  - Pašaliniai funkcijų poveikiai atskirti nuo grynų funkcijų;
  - Naudojamas atidėtasis vykdymas (angl. *lazy evaluation*);
  - Visos funkcijos grąžina reikšmes.

Būsenos nekeičiamumas

- Funkcinėse programavimo kalbose nėra kintamųjų.
- Viskas yra funkcijos.
- Haskell galima užrašyti `value = 5` — tai bus interpretuojama kaip funkcija, kuri visada grąžina 5.
- Pakartotinis priskyrimas į `value` negalimas.
- Būsenos nekeičiamumas panaikina lenktynių sąlygas: **nėra bendros modifikuojamos atminties tarp gijų**.

Grynos funkcijos

- Visos funkcijos, jei nepažymėta kitaip, yra grynos, dėl to jų kvietimą galima pakeisti grąžinama reikšme, jei funkcija kviečiama su tais pačiais parametrais.
- Jei dvi funkcijos yra grynos ir viena nenaudoja kitos rezultatų, jas visada galima vykdyti lygiagrečiai be jokių apsaugos priemonių.

Atidėtasis vykdymas

- Funkcija vykdoma tik tada, kai reikia jos rezultato.
- Funkcijos rezultato reikia tik tada, kai ji naudojama iš funkcijos, kuri nėra gryna, pvz., rašo į failą.
- Funkciniame programavime stengiamasi kuo labiau sumažinti negrynas funkcijas ir kuo daugiau kodo iškelti į grynas.
- Jei programa neturi nei vienos negrynos funkcijos, niekas nebus vykdoma.

Deklaratyvus programavimas

- Programavimas, kai nenurodome, **kaip** bus vykdoma programa, o tik **ką** reikia padaryti, vadinamas deklaratyviu.
- Atidėtasis vykdymas yra deklaratyvaus programavimo bruožas: programa nebūtinai vykdoma ta tvarka, kaip užrašyta; vykdomi nebūtinai visi sakiniai.
- Lygiagretusis programavimas visada pasako, **kaip** reikia vykdyti programą (kiek gijų sukurti, kaip paskirstyti skaičiavimus, kaip apsikeisti duomenimis).
- `1 + 2` ir 3 deklaratyvaus programavimo požiūriu nesiskiria, bet lygiagretaus — skirasi: `1 + 2` yra dar neįvykęs skaičiavimas, kurį gali reikėti vykdyti kitoje gijoje.

Concurrency vs parallelism

- *Concurrency* — kai keletas procesų ar gijų komunikuoja tarpusavyje. Tikslas dažniausiai būna suvaldyti keletą vienu metu galinčių vykti išorinių procesų.
- *Parallelism* — kai keletas procesų ar gijų vienu metu atlieka skirtingas užduotis (komunikacija nereikalinga). Tikslas dažniausiai būna pagreitinti programą.
- *Parallelism* atveju rezultatas yra deterministinis: galima nuo programuotojo paslėpti, kaip gijos komunikuoja tarpusavyje.
- *Concurrency* atveju rezultatas nėra deterministinis, o priklauso nuo to, kurios gijos suveiks pirmiau.

Esamos būsenos patikrinimas

- Haskell turi interpretatorių GHCI, kuriame galima patikrinti, kokia esama reikšmė saugoma, nekeičiant jos būsenos su komanda `sprint`.

```
Prelude> let x = 1 + 2 :: Int -- define function x
Prelude> :sprint x  -- check current state of x
x = _  -- unevaluated
Prelude> x  -- force evaluate x by printing result
3
Prelude> :sprint x  -- check x again
x = 3  -- evaluated because we printed the result above
```

Esamos būsenos patikrinimas

```
Prelude> let x = 1 + 2 :: Int
Prelude> let y = x + 3
Prelude> :sprint x
x = _
Prelude> :sprint y
y = _
Prelude> y
6
Prelude> :sprint x
x = 3
Prelude> :sprint y
y = 6
```

Esamos būsenos patikrinimas

```
Prelude Data.Tuple> let z = swap(x, x+1)
Prelude Data.Tuple> :sprint z
z = _
Prelude Data.Tuple> seq z ()
()
Prelude Data.Tuple> :sprint z
z = (_,3)
```

Esamos būsenos patikrinimas

```
Prelude> let squares = map (\x -> x * x) [1..10] :: [Int]
Prelude> :sprint squares
squares = _
Prelude> length squares
10
Prelude> :sprint squares
squares = [_,_,_,_,_,_,_,_,_,_]
```

Weak head normal form

- Funkcija seq apskaičiuoja savo argumentą, bet ne jį sudarančius komponentus.
- Toks pavidalas, kuriame yra struktūra yra apskaičiuota, bet komponentai — ne, vadinamas *weak head normal form* (*WHNF*).
- Norint lygiagrečiai paleisti skaičiavimus, reikia, kad funkcijų parametrai būtų *WHNF*, o pati funkcija juos pilnai (arba dalinai) apskaičiuotų.

`rpar` ir `rseq`

- Funkcija `rpar` priima argumentą, kuris turėtų būti skaičiuojamas lygiagrečiai.
- Funkcija `rseq` priima argumentą ir jį apskaičiuoja.
- Abi funkcijos priima argumentą ir grąžina to paties tipo atsakymą, įdėtą į `Eval` monadą.
- Monadas yra tipas, į kurį galima įdėti reikšmes ir jas kombinuoti. Pvz., `Maybe` monadas aprašo tipą, kuris gali būti be reikšmės. Tipas `Maybe Int` aprašo sveikąjį skaičių, kuris gali egzistuoti (reikšmė `Just Int`) arba ne (reikšmė `Nothing`).
- Kombinavimas vyksta perduodant funkciją, norimą atlikti su tuo, kas jau yra monade. Pvz., `Maybe Int` monadui taikant šaknies traukimo funkciją rezultatas bus `Maybe Float` tipo – monadas žino, kaip taikyti funkciją, kad esant reikšmei `Nothing` grąžintų `Nothing`, kitu atveju ištrauktų šaknį.
- `Eval` monadas skirtas į jį įdėti funkcijas, kad su jomis po to būtų galima kažką padaryti.

`rpar` ir `rseq`

- Su `Eval` monadu dirba funkcija `runEval`, kuri išima `Eval` monade esančią reikšmę.
- Kadangi monadą sukuria `rpar` ir `rseq` funkcijos, jų sukurtas monadas turi informaciją, kaip reikia apskaičiuoti jame esančią reikšmę išimant ją iš monado.
- `rpar` funkcija paleidžia skaičiavimus lygiagrečiai, bet nelaukia rezultato, o `rseq` priverstinai skaičiuoja rezultatą. Jei jis dar neskaičiuojamas, tai jis skaičiuojamas toje pačioje gijoje, jei jau skaičiuojamas dėl `rpar` panaudojimo, tai laukiama, kol skaičiavimai baigsis.
- Jei naudosime tik `rpar`, tai nebus galimybės išsiimti rezultato, jei naudosime tik `rseq`, tai skaičiavimai nebus vykdomi lygiagrečiai.

rpar ir rseq

```
someFunc = do
  let (s1, s2) = (getParallelSums [1..100000000]
                                  [100000000..200000000])
  let fullSum = s1 + s2
  print fullSum

getSquaredSum items =
  sum squares where
    squares = map (\x -> x * x) items

getParallelSums list1 list2 =
  runEval $ do
    sum1 <- rpar (getSquaredSum list1)
    sum2 <- rpar (getSquaredSum list2)
    seqSum1 <- rseq sum1
    seqSum2 <- rseq sum2
    return (seqSum1, seqSum2)
```

Strategijos

- Haskell `runEval` galima pakeisti funkcija `using`. Tokiu atveju galima nurodyti stategiją vykdymui.
- `rpar` ir `rseq` yra strategijos. Kiti strategijų pavyzdžiai:

  `rdeepseq` Pilnai apskaičiuoja savo parametrą (parametras turi realizuoti `NFData`).

  `rparWith` Apskaičiuoja parametrą lygiagrečiai pritaikant strategiją.
- Taip pat yra strategijų, kurios kitas strategijas pritaiko sąrašams (`parList`, `parListChunk` ir kt.).

Strategijos

```
someFunc :: IO ()
someFunc = do
  let numbers = getData
  let md5Hashes = (map md5 numbers `using`
                   parListChunk 100000 rpar)
  print (maximum md5Hashes)


getData :: [ByteString]
getData =
  map BLU.fromString numbersAsStrings where
    numbers = [1..1000000]
    numbersAsStrings = map show numbers
```

.NET programavimo kalbos

- .NET yra Microsoft technologija, tačiau yra atviro kodo ir veikia ne tik Windows sistemoje.
- .NET pagrindinės programavimo kalbos yra C# ir F#.
- F# yra negryna funkcinė programavimo kalba (galima perduoti nuorodas, turėti modifikuojamus kintamuosius, nenaudojamas atidėtasis vykdymas), tačiau turi gerą funkcinio programavimo palaikymą.

Parallel LINQ

- LINQ yra .NET funkcijų rinkinys, taikomas tipui `IEnumberable`.
- LINQ prideda funkcinio programavimo palaikymą .NET platformai, naudoti galima tiek C#, tiek F#.
- PLINQ yra LINQ atitkmuo, bet visos funkcijos vykdomos lygiagrečiai. LINQ ir PLINQ turi vienodus metodus, skiriasi tik vykdymo tipas.

Parallel LINQ

```
[<EntryPoint>]
let main argv =
    let numbers = seq {0..100000}
    let result = numbers.AsParallel()
                        .Where(fun n -> n % 2 = 0)
                        .Select(fun n -> n * n)
                        .Aggregate(fun acc n -> acc + n)
    printf "%d" result
    0
```

PSeq

- Natūralesnis funkciniam programavimui sprendimas yra `PSeq` iš `FSharp.Collections.ParallelSeq` paketo.
- Turi panašias funkcijas, kaip PLINQ, bet galima taikyti su `|>` operatoriumi.
- `|>` operatorius taikomas dviems operandams, pirmasis yra reikšmė, antrasis — funkcija. Reikšmė perduodama parametru funkcijai, tokiu būdu galima tvarkingai užrašyti, kokias duomenų transformacijas reikia atlikti duomenims, kad gautume rezultatą.

PSeq

```
[<EntryPoint>]
let main argv =
    let numbers = seq {0..100000}
    let result = numbers |> PSeq.filter(fun n -> n % 2 = 0)
                         |> PSeq.map(fun n -> n * n)
                         |> PSeq.reduce(fun acc n -> acc + n)

    printf "%d" result
    0
```

Reikalavimai aukštesnės eilės funkcijoms

- Naudotis lygiagrečiaisiais `map`, `filter` ir `reduce` galima daugeliu atvejų, tačiau nevisais.
- `map` ir `filter` lygiagretinti galima tada, kai parametru paduodamos funkcijos gali būti vykdomos lygiagrečiai ir nesukelti lenktynių sąlygų. Daugeliu atvejų tai būna grynos funkcijos, bet nebūtinai turi būti tokios.
- `reduce` funkcijoms kyla papildomas reikalavimas: jos turėtų būti monoidai.

reduce ir monoidai

- Monoidas yra funkcija, kuri tenkina keletą kriterijų:
  - Parametrai ir grąžinamas rezultatas yra to paties tipo;
  - Funkcija yra asociatyvi, t. y., `func(a, (func(b, c))) == func(func(a, b), c)`;
  - Nekeisti grąžinamo tipo priklausomai nuo suskaičiuoto rezultato;
  - Turėti identiteto elementą, t. y., tokią reikšmę, kuri nemodifikuoja pirmosios,
- Sveikųjų skaičių sumos funkcija yra monoidas, nes parametrai ir rezultatas yra `int` tipo, `a + (b + c) == (a + b) + c`, rezultato tipas visada toks pat bei reikšmė 0 yra identiteto elementas: `a + 0 == a`.

Haskell gijos

- Haskell kiekviena funkcija turi grąžinti kokio nors tipo reikšmę.
- Imperatyvaus programavimo kalbose gijos paleidimo funkcija niekada negrąžina jokios reikšmės.
- Jei Haskell reikia atlikti kokius nors veiksmums gijoje, bet nieko negrąžinti, naudojama funkcija `forkIO`.
- Visi įvesties / išvesties veiksmai Haskell grąžina `IO` tipo reikšmę, taip pat ir `forkIO`.

Haskell gijos

```haskell
someFunc :: IO ()
someFunc = do
  forkIO (execute "First")
  forkIO (execute "Second")
  execute "Third"

execute :: String -> IO ()
execute name = do
  putStrLn (name ++ ": one")
  putStrLn (name ++ ": two")
  putStrLn (name ++ ": three")
```

Komunikacija tarp gijų

- Gijoms apsikeisti duomenimis gali būti naudojamas `MVar`.
- `MVar` galima įsivaizduoti kaip dėžutę, kurioje reikšmė gali būti arba nebūti. Reikšmę įdėti galima su funkcija `putMVar`, išimti su `takeMVar`.
- Abi funkcijos dirba `IO` monade, t. y., abi operacijos turi šalutinius poveikius.
- Abi funkcijos gali būti blokuojančios, kai norima įdėti į užimtą `MVar` arba išimti iš tuščio.

Komunikacija tarp gijų

```haskell
data Counter = Counter (MVar Int)

initCounter :: IO Counter
initCounter = do
  m <- newMVar 0
  let counter = Counter m
  return counter

changeCounter :: Counter -> Int -> IO ()
changeCounter (Counter m) modifier = do
  counterValue <- takeMVar m
  putMVar m (counterValue + modifier)

getCounterValue :: Counter -> IO Int
getCounterValue (Counter m) = do
  counterValue <- takeMVar m
  return counterValue
```

Komunikacija tarp gijų

```haskell
runIncreaser :: Counter -> MVar () -> IO ()
runIncreaser counter waiter = do
  forM_ [1..50] $ \i -> do
    changeCounter counter 1
  putMVar waiter ()

runDecreaser :: Counter -> MVar () -> IO ()
runDecreaser counter waiter = do
  forM_ [1..50] $ \i -> do
    changeCounter counter (-1)
  putMVar waiter ()
```

Komunikacija tarp gijų

```haskell
someFunc :: IO ()
someFunc = do
  waiter <- newEmptyMVar
  counter <- initCounter
  forM_ [1..10] $ \i -> do
    forkIO (runIncreaser counter waiter)
  forM_ [11..19] $ \i -> do
    forkIO (runDecreaser counter waiter)
  forM_ [1..19] $ \i -> do
    takeMVar waiter
    return ()
  counterValue <- getCounterValue counter
  print counterValue
```

Sinchronizacija Haskell

- Kadangi visa modifikuojama būsena yra įdėta į `MVar` monadą, kuris pats yra sinchronizuotas, o programa gali dirbti su reikšme tik tada, kai ji išimta, papildomų sinchronizacijos priemonių nereikia.
- Jei Haskell reikia sinchronizacijos (pvz., kelioms gijoms spausdinant vienu metu į ekraną), sinchronizacijai taip pat naudojamas `MVar`.