Parallel Functional Programming

Andrés Sicard-Ramírez

Ciclo de Conferencias Apolo
Universidad EAFIT
2017-09-27

Preliminary Concepts

Parallel computing

> *"Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously." [Wikipedia 2017-09-27]*

Preliminary Concepts

Parallel computing

"Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously." [Wikipedia 2017-09-27]

Functional programming

"In computer science, functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data." [Wikipedia 2017-09-27]

Preliminary Concepts

Side effects

> *"A side effect introduces a dependency between the global state of the system and the behaviour of a function... Side effects are essentially invisible inputs to, or outputs from, functions."* *

*O'Sullivan, B., Goerzen, J. and Stewart, D. (2008). Real World Haskell, p. 27.

†Hutton, G. (2007). Programming in Haskell, p. 87.

Preliminary Concepts

Side effects

> *"A side effect introduces a dependency between the global state of the system and the behaviour of a function... Side effects are essentially invisible inputs to, or outputs from, functions."* *

Pure functions

Pure functions *"take all their input as explicit arguments, and produce all their output as explicit results."* †

*O'Sullivan, B., Goerzen, J. and Stewart, D. (2008). Real World Haskell, p. 27.
†Hutton, G. (2007). Programming in Haskell, p. 87.

Preliminary Concepts

Side effects

> *"A side effect introduces a dependency between the global state of the system and the behaviour of a function... Side effects are essentially invisible inputs to, or outputs from, functions."* [*]

Pure functions

Pure functions *"take all their input as explicit arguments, and produce all their output as explicit results."* [†]

Referential transparency

Equals can be replaced by equals.

[*] O'Sullivan, B., Goerzen, J. and Stewart, D. (2008). Real World Haskell, p. 27.
[†] Hutton, G. (2007). Programming in Haskell, p. 87.

Preliminary Concepts

Evaluation of expressions: Strict and non-strict languages

In non-strict languages nothing is evaluated until necessary.

*http://www.cs.nott.ac.uk/~pszgmh/pgp.html .

Preliminary Concepts

Evaluation of expressions: Strict and non-strict languages

In non-strict languages nothing is evaluated until necessary.

Example

See whiteboard.

*http://www.cs.nott.ac.uk/~pszgmh/pgp.html .

Preliminary Concepts

Evaluation of expressions: Strict and non-strict languages

In non-strict languages nothing is evaluated until necessary.

Example

See whiteboard.

Lazy evaluation = non-strict evaluation + sharing

See Hutton slides: Chapter 15 - Lazy evaluation.*

*http://www.cs.nott.ac.uk/~pszgmh/pgp.html .

Preliminary Concepts

Definition (Weak head normal form)

A λ-expression is in weak head normal (WHFN) form if and only it is the form

$$F\, E_1\, E_2 \dots E_n \text{ where } n \geq 0;$$

and either F is a variable or data object or F is a λ-abstraction or built-in function and $(F\, E_1\, E_2 \dots E_m)$ is not a redex for any $m \leq n$.
An expression has no top-level redex if and only if it is in weak head normal form.*

Example

The following expressions are in WHFN: 42, $(+)((-)\ 4\ 3)$, $\lambda x.5 + 1$ and $\mathrm{cons}\ x\ xs$.

*Peyton Jones, S. L. (1987). The Implementation of Functional Programming Languages, p. 198.

Pure Functional Languages

Examples

- AGDA (proof-assistant) (version 2.5.3, September 2017)
- CLEAN (version 2.4, December 2011)
- COQ (proof-assistant) (version 8.6.1, July 2017)
- CURRY (functional-logic) (KICS2 version 0.6.0-b6, July 2017)
- FREGE (functional-logic) (version 3.24, March 2016)
- HASKELL (GHC version 8.2.1, July 2017)
- HOPE (developed in the 1970s)
- IDRIS (proof-assistant) (version 1.1.1, August 5, 2017)
- MERCURY (functional-logic) (version 14.01.1, September 2014)
- MIRANDA (developed in the 1980s)

Parallel Computing in Haskell[†]

[*]Marlow, S. (2012). Parallel and Concurrent Programming in Haskell, p. 342.

[†]The Haskell examples shown in these slides are from [Marlow 2013]. These examples were tested with GHC 8.2.1 and the PARALLEL 3.2.1.1 library.

Parallel Computing in Haskell[†]

- Parallel programming in Haskell is deterministic
  - If a parallel program gives a result it always is the same.

[*]Marlow, S. (2012). Parallel and Concurrent Programming in Haskell, p. 342.
[†]The Haskell examples shown in these slides are from [Marlow 2013]. These examples were tested with GHC 8.2.1 and the PARALLEL 3.2.1.1 library.

Parallel Computing in Haskell[†]

- Parallel programming in Haskell is deterministic
  - If a parallel program gives a result it always is the same.
  - *"Deterministic parallel programming is the best of both worlds: testing, debugging and reasoning can be performed on the sequential program, but the program runs faster when processors are added."* [*]

[*] Marlow, S. (2012). Parallel and Concurrent Programming in Haskell, p. 342.

[†] The Haskell examples shown in these slides are from [Marlow 2013]. These examples were tested with GHC 8.2.1 and the PARALLEL 3.2.1.1 library.

Parallel Computing in Haskell

- Parallel Haskell programs do not explicitly deal with synchronisation or communication
  - *"Synchronisation is the act of waiting for other tasks to complete, perhaps due to data dependencies. Communication involves the transmission of results between tasks running on different processors. Synchronisation is handled automatically by the GHC runtime system and/or the parallelism libraries. Communication is implicit in GHC since all tasks share the same heap, and can share objects without restriction."* *

*Marlow, S. (2012). Parallel and Concurrent Programming in Haskell, p. 343.

†Marlow, S. (2013). Parallel and Concurrent Programming in Haskell, p. 6.

Parallel Computing in Haskell

- Parallel Haskell programs do not explicitly deal with synchronisation or communication
  - *"Synchronisation is the act of waiting for other tasks to complete, perhaps due to data dependencies. Communication involves the transmission of results between tasks running on different processors. Synchronisation is handled automatically by the GHC runtime system and/or the parallelism libraries. Communication is implicit in GHC since all tasks share the same heap, and can share objects without restriction."* *
  - *"This is both a blessing and a curse."* †

*Marlow, S. (2012). Parallel and Concurrent Programming in Haskell, p. 343.
†Marlow, S. (2013). Parallel and Concurrent Programming in Haskell, p. 6.

Eval, rpar and rseq

The `Eval` data type

`Eval` is a polymorphic data type that makes it easier to define parallel strategies.

```
data Eval a = ...
```

Eval, rpar and rseq

The `Eval` data type

`Eval` is a polymorphic data type that makes it easier to define parallel strategies.

```
data Eval a = ...
```

The `Eval` monad

`Eval` is a strict identity monad, that is, in `m >>= f`, the computation `m` is evaluated before the result is passed to `f`.

```
instance Monad Eval where ...
```

Eval, rpar and rseq

Running the computation

The `runEval` function performs the `Eval` computation and returns its result.

```
runEval :: Eval a → a
```

Eval, rpar and rseq

The rpar and rseq combinators

> *"The* `rpar` *combinator is used for creating parallelism; it says "my argument could be evaluated in parallel", while* `rseq` *is used for forcing sequential evaluation: it says "evaluate my argument now" (to weak-head normal form)."* *

```
rpar :: a → Eval a
rseq :: a → Eval a
```

*Marlow, S. (2012). Parallel and Concurrent Programming in Haskell, p. 346.

Examples using rpar and rseq

Let `f` be a function and suppose that `f x` takes longer to evaluate than `f y`.

Example (`rpar`/`rpar`)

```
runEval $ do
  fx ← rpar (f x)
  fy ← rpar (f y)
  return (fx, fy)
```

Running behaviour: See [Marlow 2013, Figure 2.5].

Examples using rpar and rseq

Let `f` be a function and suppose that `f x` takes longer to evaluate than `f y`.

Example (`rpar`/`rseq`)

```
runEval $ do
  fx ← rpar (f x)
  fy ← rseq (f y)
  return (fx, fy)
```

Running behaviour: See [Marlow 2013, Figure 2.6].

Examples using rpar and rseq

Let `f` be a function and suppose that `f x` takes longer to evaluate than `f y`.

Example (`rpar`/`rpar`/`rseq`/`rseq`)

```
runEval $ do
  fx ← rpar (f x)
  fy ← rpar (f y)
  rseq fx
  rseq fy
  return (fx, fy)
```

Running behaviour: See [Marlow 2013, Figure 2.7].

Example: Paralleling a Sudoku Solver

The solve function

```
solve :: String → Maybe Grid
```

Example (sudoku1.hs)

```
main :: IO ()
main = do
  [f]  ← getArgs
  file ← readFile f

  let puzzles :: [String]
      puzzles = lines file

      solutions :: [Maybe Grid]
      solutions = map solve puzzles

  print (length (filter isJust solutions))
```

Example: Paralleling a Sudoku Solver

Example (sudoku2.hs)

```
main :: IO ()
main = do
  [f]  ← getArgs
  file ← readFile f

  let puzzles :: [String]
      puzzles = lines file
  ...
```

Example: Paralleling a Sudoku Solver

Example (sudoku2.hs (cont.))

```
    ...
    let as, bs :: [String]
        (as,bs) = splitAt (length puzzles `div` 2) puzzles

        solutions :: [Maybe Grid]
        solutions = runEval $ do
                      as' ← rpar (force (map solve as))
                      bs' ← rpar (force (map solve bs))
                      rseq as'
                      rseq bs'
                      return (as' ++ bs')

    print (length (filter isJust solutions))
```

Example: Paralleling a Sudoku Solver

Evaluating to normal form

```
force :: NFData a → a → a
```

Static/Dynamic partitioning

```
parMap :: (a → b) → [a] → Eval [b]
parMap f [] = return []
parMap f (a:as) = do
  b  ← rpar (f a)
  bs ← parMap f as
  return (b:bs)
```

Example: Paralleling a Sudoku Solver

Example (sudoku3.hs)

```
main :: IO ()
main = do
  [f]  ← getArgs
  file ← readFile f

  let puzzles :: [String]
      puzzles = lines file

      solutions :: [Maybe Grid]
      solutions = runEval (parMap solve puzzles)

  print (length (filter isJust solutions))
```

Evaluation Strategies for Parallel Haskell

Evaluation Strategies

Evaluation strategies are an import abstraction for adding pure, deterministic, parallelism to Haskell programs. They were initially designed in 2002* and redesigned in 2010.†

*Trinder, P. W., Loidl, H.-W. and Pointon, R. F. (2002). Parallel and Distributed Haskells.

†Marlow, S., Maier, P., Loidl, H.-W., Aswad, M. K. and Trinder, P. (2010). Seq No More: Better Strategies for Parallel Haskell.

Evaluation Strategies for Parallel Haskell

Evaluation Strategies

Evaluation strategies are an import abstraction for adding pure, deterministic, parallelism to Haskell programs. They were initially designed in 2002* and redesigned in 2010.†

The strategy type

```
type Strategy a = a → Eval a
```

*Trinder, P. W., Loidl, H.-W. and Pointon, R. F. (2002). Parallel and Distributed Haskells.

†Marlow, S., Maier, P., Loidl, H.-W., Aswad, M. K. and Trinder, P. (2010). Seq No More: Better Strategies for Parallel Haskell.

References

Hutton, G. (2007). Programming in Haskell. Cambridge University Press.

Marlow, S. (2012). Parallel and Concurrent Programming in Haskell. In: Central European Functional Programming School (CEFP 2011). Ed. by Zsók, V., Horváth, Z. and Plasmeijer, R. Vol. 7241. Lecture Notes in Computer Science, pp. 333–401. DOI: 10.1007/978-3-642-32096-5_7.

— (2013). Parallel and Concurrent Programming in Haskell. O'Reilly Media, Inc.

Marlow, S., Maier, P., Loidl, H.-W., Aswad, M. K. and Trinder, P. (2010). Seq No More: Better Strategies for Parallel Haskell. In: Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10), pp. 91–102. DOI: 10.1145/1863523.1863535.

O'Sullivan, B., Goerzen, J. and Stewart, D. (2008). Real World Haskell. O'REILLY.

Peyton Jones, S. L. (1987). The Implementation of Functional Programming Languages. Prentice-Hall International.

Trinder, P. W., Loidl, H.-W. and Pointon, R. F. (2002). Parallel and Distributed Haskells. Journal of Functional Programming 12.4–5, pp. 469–510. DOI: 10.1017/S0956796802004343.