Entropiafix für nebenleufiges in Hasskel

Frederick Bullik

6. Juli 2008

Inhaltsverzeichnis

1 Einleitung

2 States
- Erklärung
- Beispiel

3 Monaden
- Funktionsweise
- Nebenläufigkeit

4 Software Transaktional Memeory

5 Zusammenfassung

6 Das Letzte . . .

Inhaltsverzeichnis

1. Einleitung
2. States
   - Erklärung
   - Beispiel
3. Monaden
   - Funktionsweise
   - Nebenläufigkeit
4. Software Transaktional Memeory
5. Zusammenfassung
6. Das Letzte . . .

Inhaltsverzeichnis

1. Einleitung
2. States
   - Erklärung
   - Beispiel
3. Monaden
   - Funktionsweise
   - Nebenläufigkeit
4. Software Transaktional Memeory
5. Zusammenfassung
6. Das Letzte . . .

Inhaltsverzeichnis

1. Einleitung
2. States
   - Erklärung
   - Beispiel
3. Monaden
   - Funktionsweise
   - Nebenläufigkeit
4. Software Transaktional Memeory
5. Zusammenfassung
6. Das Letzte . . .

Inhaltsverzeichnis

1. Einleitung
2. States
   - Erklärung
   - Beispiel
3. Monaden
   - Funktionsweise
   - Nebenläufigkeit
4. Software Transaktional Memeory
5. Zusammenfassung
6. Das Letzte . . .

Inhaltsverzeichnis

1. Einleitung
2. States
   - Erklärung
   - Beispiel
3. Monaden
   - Funktionsweise
   - Nebenläufigkeit
4. Software Transaktional Memeory
5. Zusammenfassung
6. Das Letzte . . .

Überblick

1. Einleitung
2. States
   - Erklärung
   - Beispiel
3. Monaden
   - Funktionsweise
   - Nebenläufigkeit
4. Software Transaktional Memeory
5. Zusammenfassung
6. Das Letzte ...

Warum nebenläfig

- Prozessoren werden nicht scheneller ...
- Aber sie werden mehr.

Warum nebenläfig

- Prozessoren werden nicht scheneller . . .
- Aber sie werden mehr.

Warum Haskell

- Pure Implementation
- Unterstüzt Nebenläufigkeit
- Wird aktiv entwickelt

Warum Haskell

- Pure Implementation
- Unterstüzt Nebenläufigkeit
- Wird aktiv entwickelt

Warum Haskell

- Pure Implementation
- Unterstüzt Nebenläufigkeit
- Wird aktiv entwickelt

Überblick

1. Einleitung
2. States
   - Erklärung
   - Beispiel
3. Monaden
   - Funktionsweise
   - Nebenläufigkeit
4. Software Transaktional Memeory
5. Zusammenfassung
6. Das Letzte . . .

Kein state

- Objektorientiert:
  Verarbeiten von Variablen
  Sorgen um deren State
- Funktional:
  Nur Konstanten
  Funktionen geben immer selbes Ergebnis zurück

Kein state

- Objektorientiert:
  Verarbeiten von Variablen
  Sorgen um deren State
- Funktional:
  Nur Konstanten
  Funktionen geben immer selbes Ergebnis zurück

Nebenläufigkeit ?

Was hat das mit Nebenläufigkeit zu tun?

- Eine Funktion gibt immer einen festen Wert zurück.
- $\Rightarrow$ d.h verteilbar

Nebenläufigkeit ?

Was hat das mit Nebenläufigkeit zu tun?

- Eine Funktion gibt immer einen festen Wert zurück.
- $\Rightarrow$ d.h verteilbar

Objektorientiert ⇒ Call by Reference

Seiteneffekt! State!

```
void Triple(int& x)
        {
                x *= 3;
        }
```

Funktional $\Rightarrow$ Call by Need

Beispiel: Berechnen der Fibonacci-Zahlen

$$fib :: Integer \rightarrow Integer$$
$$fib\ 0 = 0$$
$$fib\ 1 = 1$$
$$fib\ n = (fib\ (n-1)) + (fib\ (n-2))$$

Überblick

1. Einleitung
2. States
   - Erklärung
   - Beispiel
3. Monaden
   - Funktionsweise
   - Nebenläufigkeit
4. Software Transaktional Memeory
5. Zusammenfassung
6. Das Letzte . . .

Warum Monaden

- nur Funktionen ohne Seitenefekte == Backofen
- Funktionen nicht zeitlich geordnet ⇒ Deadlocks
- Wir wollen Seiteneffekte, aber funktional bleiben.
- Lösung: Monaden

Warum Monaden

- nur Funktionen ohne Seitenefekte == Backofen
- Funktionen nicht zeitlich geordnet $\Rightarrow$ Deadlocks
- Wir wollen Seiteneffekte, aber funktional bleiben.
- Lösung: Monaden

Warum Monaden

- nur Funktionen ohne Seitenefekte == Backofen
- Funktionen nicht zeitlich geordnet ⇒ Deadlocks
- Wir wollen Seiteneffekte, aber funktional bleiben.
- Lösung: Monaden

Warum Monaden

- nur Funktionen ohne Seitenefekte == Backofen
- Funktionen nicht zeitlich geordnet ⇒ Deadlocks
- Wir wollen Seiteneffekte, aber funktional bleiben.
- Lösung: Monaden

Don't Panic

- Monaden kommen aus der Kategorien-Theorie
- Monaden sind eigentlich warme, kuschelige Dinger.

Don't Panic

- Monaden kommen aus der Kategorien-Theorie
- Monaden sind eigentlich warme, kuschelige Dinger.

- Kann man als eine Art Aktion sehen[1]
- Type IO $a = World \rightarrow (a, World)$
- ">>=" klebt Aktionen zusammen $\Rightarrow$ do
- kapseln Seiteneffekte

[1] Das Typsystem kümmert sich um den State

- Kann man als eine Art Aktion sehen[1]
- Type IO $a = World \rightarrow (a, World)$
- ">>=" klebt Aktionen zusammen $\Rightarrow$ do
- kapseln Seiteneffekte

[1] Das Typsystem kümmert sich um den State

- Kann man als eine Art Aktion sehen[1]
- Type IO $a = World \rightarrow (a, World)$
- ">>=" klebt Aktionen zusammen $\Rightarrow$ do
- kapseln Seiteneffekte

[1] Das Typsystem kümmert sich um den State

- Kann man als eine Art Aktion sehen[1]
- Type IO $a = World \rightarrow (a, World)$
- ">>=" klebt Aktionen zusammen $\Rightarrow$ do
- kapseln Seiteneffekte

[1] Das Typsystem kümmert sich um den State

Der State, der durch die Monade gekapselt wird, mußnicht single threaded sein.

Überblick

1. Einleitung
2. States
   - Erklärung
   - Beispiel
3. Monaden
   - Funktionsweise
   - Nebenläufigkeit
4. Software Transaktional Memeory
5. Zusammenfassung
6. Das Letzte . . .

forkIO

MVar

QSem

Überblick

1. Einleitung
2. States
   - Erklärung
   - Beispiel
3. Monaden
   - Funktionsweise
   - Nebenläufigkeit
4. Software Transaktional Memeory
5. Zusammenfassung
6. Das Letzte . . .

Warum

- keine Logs
  $\Rightarrow$ das bedeutet
  - kein Overlocking
  - kein Underlocking
  - keine Deadlocks
  - ...

Warum

- keine Logs
  $\Rightarrow$ das bedeutet
  - kein Overlocking
  - kein Underlocking
  - keine Deadlocks
  - . . .

Warum

- keine Logs
  $\Rightarrow$ das bedeutet
  - kein Overlocking
  - kein Underlocking
  - keine Deadlocks
  - ...

Warum

- keine Logs

  $\Rightarrow$ das bedeutet
  - kein Overlocking
  - kein Underlocking
  - keine Deadlocks
  - ...

Warum

- keine Logs
  $\Rightarrow$ das bedeutet
  - kein Overlocking
  - kein Underlocking
  - keine Deadlocks
  - . . .

Funktionsweise

- Wrappen der Funktionen in einen Atomic-block
- Funktionen sind isoliert
- Entweder alles oder nichts
- Wenn Fail $\Rightarrow$ Retry
- System kümmert sich um das Retry
- orElse $\Leftarrow$ Was führen wir bei einem Retry aus?

Funktionsweise

- Wrappen der Funktionen in einen Atomic-block
- Funktionen sind isoliert
- Entweder alles oder nichts
- Wenn Fail $\Rightarrow$ Retry
- System kümmert sich um das Retry
- orElse $\Leftarrow$ Was führen wir bei einem Retry aus?

Funktionsweise

- Wrappen der Funktionen in einen Atomic-block
- Funktionen sind isoliert
- Entweder alles oder nichts
- Wenn Fail $\Rightarrow$ Retry
- System kümmert sich um das Retry
- orElse $\Leftarrow$ Was führen wir bei einem Retry aus?

Funktionsweise

- Wrappen der Funktionen in einen Atomic-block
- Funktionen sind isoliert
- Entweder alles oder nichts
- Wenn Fail $\Rightarrow$ Retry
- System kümmert sich um das Retry
- orElse $\Leftarrow$ Was führen wir bei einem Retry aus?

Funktionsweise

- Wrappen der Funktionen in einen Atomic-block
- Funktionen sind isoliert
- Entweder alles oder nichts
- Wenn Fail $\Rightarrow$ Retry
- System kümmert sich um das Retry
- orElse $\Leftarrow$ Was führen wir bei einem Retry aus?

Funktionsweise

- Wrappen der Funktionen in einen Atomic-block
- Funktionen sind isoliert
- Entweder alles oder nichts
- Wenn Fail $\Rightarrow$ Retry
- System kümmert sich um das Retry
- orElse $\Leftarrow$ Was führen wir bei einem Retry aus?

Überblick

1. Einleitung
2. States
   - Erklärung
   - Beispiel
3. Monaden
   - Funktionsweise
   - Nebenläufigkeit
4. Software Transaktional Memeory
5. Zusammenfassung
6. Das Letzte ...

Literatur

Simon Peyton Jones, Andrew Gorden, Sigbjorn Finne
Concurrent Haskell, 1996

Kevin Hammond
Parrallel Functional Programming: An Introduction

Simon Peyton Jones
Tackeling the Awkward Squad: monadic input/output, concurrency, exceptions, foreign-language calls in Haskell, 2008

Joe Armstrong
Programming Erlang - Software for a Concurrent World
The Pragmatic Programmers, 2007

Literatur

Simon Peyton Jones, Andrew Gorden, Sigbjorn Finne
Concurrent Haskell, 1996

Kevin Hammond
Parrallel Functional Programming: An Introduction

Simon Peyton Jones
Tackeling the Awkward Squad: monadic input/output, concurrency, exceptions, foreign-language calls in Haskell, 2008

Joe Armstrong
Programming Erlang - Software for a Concurrent World
The Pragmatic Programmers, 2007

Literatur

- Simon Peyton Jones, Andrew Gorden, Sigbjorn Finne
  Concurrent Haskell, 1996
- Kevin Hammond
  Parrallel Functional Programming: An Introduction
- Simon Peyton Jones
  Tackeling the Awkward Squad: monadic input/output, concurrency, exceptions, foreign-language calls in Haskell, 2008
- Joe Armstrong
  Programming Erlang - Software for a Concurrent World
  The Pragmatic Programmers, 2007

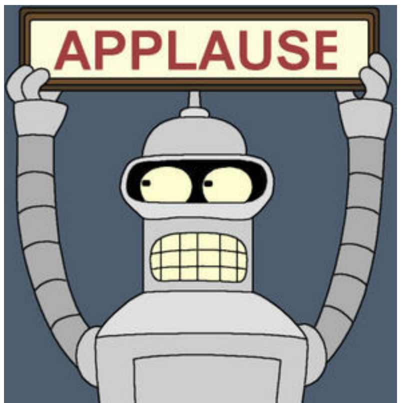
Literatur

- Simon Peyton Jones, Andrew Gorden, Sigbjorn Finne
  Concurrent Haskell, 1996
- Kevin Hammond
  Parrallel Functional Programming: An Introduction
- Simon Peyton Jones
  Tackeling the Awkward Squad: monadic input/output, concurrency, exceptions, foreign-language calls in Haskell, 2008
- Joe Armstrong
  Programming Erlang - Software for a Concurrent World
  The Pragmatic Programmers, 2007

Fragen

Noch Fragen?

Danke



Danke für die Aufmerksamkeit