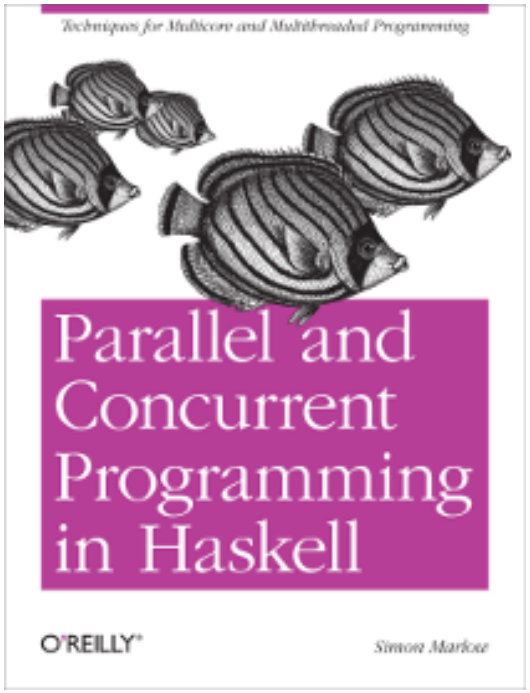
Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

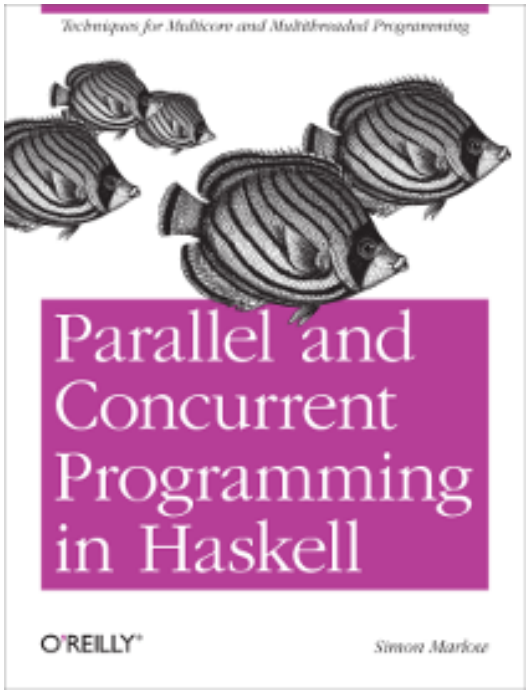
Übersicht I

Leseempfehlung:



Wunderbares Buch zum Thema von Simon Marlow.

Leseempfehlung:



Wunderbares Buch zum Thema von Simon Marlow.

Nicht in der Uni-Bibliothek, dafür aber Gratis im Internet verfügbar, inklusive Beispielcode auf `Hackage`.

Motivation

Free Lunch is over!

Herb Sutter (2005)

Free Lunch is over!

Herb Sutter (2005)

Die Hardware unserer Computer wird seit mehreren Jahren schon schneller breiter (*mehr* Kerne) als tiefer (*schnellere* Kerne).

Free Lunch is over!

Herb Sutter (2005)

Die Hardware unserer Computer wird seit mehreren Jahren schon schneller breiter (*mehr* Kerne) als tiefer (*schnellere* Kerne).

Um technischen Fortschritt voll auszunutzen ist es also essentiell, gute Werkzeuge für einfache und effiziente Parallelisierung bereit zu stellen.

Definitionen

Parallelism vs. Concurrency:

Beides ist ein Ausdruck von „Dinge gleichzeitig tun“; in der Programmierung haben sie aber grundverschiedene Bedeutungen.

Parallelism vs. Concurrency:

Beides ist ein Ausdruck von „Dinge gleichzeitig tun“; in der Programmierung haben sie aber grundverschiedene Bedeutungen.

Programme arbeiten *parallel*, wenn sie mehrere Prozessorkerne einsetzen, um schneller an die Antwort einer bestimmten Frage zu kommen.

Parallelism vs. Concurrency:

Beides ist ein Ausdruck von „Dinge gleichzeitig tun“; in der Programmierung haben sie aber grundverschiedene Bedeutungen.

Programme arbeiten *parallel*, wenn sie mehrere Prozessorkerne einsetzen, um schneller an die Antwort einer bestimmten Frage zu kommen.

Nebenläufige Programme hingegen haben mehrere „threads of control“. Oft dient das dazu, gleichzeitig mit mehreren externen Agenten (dem User, einer Datenbank, . . .) zu interagieren.

More foo about parallelism and determinism and such...

(WH)NF:

Im Themenbereich Parallelism wird oft darüber gesprochen, wann Ausdrücke ausgewertet werden und „wie weit“ (Laziness). Es gibt dafür zwei wichtige Vokabeln: **Normal Form** und **Weak Head Normal Form**.

(WH)NF:

Im Themenbereich Parallelism wird oft darüber gesprochen, wann Ausdrücke ausgewertet werden und „wie weit“ (Laziness). Es gibt dafür zwei wichtige Vokabeln: **Normal Form** und **Weak Head Normal Form**.

Die **NF** eines Ausdrucks ist der gesamte Ausdruck, vollständig berechnet. Es gibt keine Unterausdrücke, die weiter ausgewertet werden könnten.

(WH)NF:

Im Themenbereich Parallelism wird oft darüber gesprochen, wann Ausdrücke ausgewertet werden und „wie weit“ (Laziness). Es gibt dafür zwei wichtige Vokabeln: **Normal Form** und **Weak Head Normal Form**.

Die **NF** eines Ausdrucks ist der gesamte Ausdruck, vollständig berechnet. Es gibt keine Unterausdrücke, die weiter ausgewertet werden könnten.

Die **WHNF** eines Ausdrucks ist der Ausdruck, evaluiert zum äußersten Konstruktor oder zur äußersten λ-Abstraktion (dem *head*). Unterausdrücke können berechnet sein oder auch nicht. Ergo ist jeder Ausdruck in **NF** auch in **WHNF**.

(WH)NF Zuschauer-Wachzustands-Überprüfungs-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

(WH)NF Zuschauer-Wachzustands-Überprüfungs-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

(WH)NF Zuschauer-Wachzustands-Überprüfungs-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

$\Rightarrow$ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

(WH)NF Zuschauer-Wachzustands-Überprüfungs-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

$\Rightarrow$ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```
\x -> 2 + 2
```

(WH)NF Zuschauer-Wachzustands-Überprüfungs-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

⇒ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```
\x -> 2 + 2
```

⇒ **WHNF**! Der *head* ist eine λ-Abstraktion.

(WH)NF Zuschauer-Wachzustands-Überprüfungs-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

⇒ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```
\x -> 2 + 2
```

⇒ **WHNF**! Der *head* ist eine λ-Abstraktion.

```
'f' : ("oo" ++ "bar")
```

(WH)NF Zuschauer-Wachzustands-Überprüfungs-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

⇒ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```
\x -> 2 + 2
```

⇒ **WHNF**! Der *head* ist eine λ-Abstraktion.

```
'f' : ("oo" ++ "bar")
```

⇒ **WHNF**! Der *head* ist der Konstruktor `(:)`.

(WH)NF Zuschauer-Wachzustands-Überprüfungs-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

⇒ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```
\x -> 2 + 2
```

⇒ **WHNF**! Der *head* ist eine λ-Abstraktion.

```
'f' : ("oo" ++ "bar")
```

⇒ **WHNF**! Der *head* ist der Konstruktor `(:)`.

```
(\x -> x + 1) 2
```

(WH)NF Zuschauer-Wachzustands-Überprüfungs-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

⇒ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```
\x -> 2 + 2
```

⇒ **WHNF**! Der *head* ist eine λ-Abstraktion.

```
'f' : ("oo" ++ "bar")
```

⇒ **WHNF**! Der *head* ist der Konstruktor `(:)`.

```
(\x -> x + 1) 2
```

⇒ Weder noch! Äußerster Part ist Anwendung der Funktion.

Ein paar technische Feinheiten:

Ein paar technische Feinheiten:

Um Programme in `Haskell` parallel ausführen zu können, müssen sie wie folgt kompiliert werden:

```
$ ghc -make -rtsopts -threaded Main.hs
```

Ein paar technische Feinheiten:

Um Programme in `Haskell` parallel ausführen zu können, müssen sie wie folgt kompiliert werden:

```
$ ghc -make -rtsopts -threaded Main.hs
```

Danach können sie auch mit RTS (Run Time System) - Optionen wie z.B. diesen hier ausgeführt werden:

```
$ ./Main.hs +RTS -N2 -s -RTS
```

Ein paar technische Feinheiten:

Um Programme in `Haskell` parallel ausführen zu können, müssen sie wie folgt kompiliert werden:

```
$ ghc -make -rtsopts -threaded Main.hs
```

Danach können sie auch mit RTS (Run Time System) - Optionen wie z.B. diesen hier ausgeführt werden:

```
$ ./Main.hs +RTS -N2 -s -RTS
```

Dokumentation findet sich leicht via beliebiger Suchmaschine.
Eine Kurzübersicht gibt es zum Beispiel unter
`cheatography.com/nash/cheat-sheets/ghc-and-rts-options/`

Parallelism

- Die `Eval`-Monade und Strategies
- Die `Par`-Monade
- Die `RePa`-Bibliothek
- GPU-Programming mit `Accelerate`

Parallelism

- ○ Die `Eval`-Monade und Strategies
- Die `Par`-Monade
- Die `RePa`-Bibliothek
- GPU-Programming mit `Accelerate`

Das Modul `Control.Parallel.Strategies` (aus dem Paket `parallel`) stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung, ...

Das Modul `Control.Parallel.Strategies` (aus dem Paket `parallel`) stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung, ...

```
type Strategy a = a -> Eval a
```

Das Modul `Control.Parallel.Strategies` (aus dem Paket `parallel`) stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung, ...

```
type Strategy a = a -> Eval a
```

... insbesondere die Strategies `rpar` und `rseq`. Dazu gleich mehr.

Das Modul `Control.Parallel.Strategies` (aus dem Paket `parallel`) stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung, ...

```
type Strategy a = a -> Eval a
```

... insbesondere die Strategies `rpar` und `rseq`. Dazu gleich mehr.

Desweiteren stellt es die Operation `runEval`, die die monadischen Berechnungen ausführt und das Ergebnis zurück gibt, bereit.

```
runEval :: Eval a -> a
```

Das Modul `Control.Parallel.Strategies` (aus dem Paket `parallel`) stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung, ...

```
type Strategy a = a -> Eval a
```

... insbesondere die Strategies `rpar` und `rseq`. Dazu gleich mehr.

Desweiteren stellt es die Operation `runEval`, die die monadischen Berechnungen ausführt und das Ergebnis zurück gibt, bereit.

```
runEval :: Eval a -> a
```

Wohlgemerkt: `runEval` ist *pur!*
Wir müssen nicht gleichzeitig auch in der `IO`-Monade sein.

`rpar` ist die Strategie, die ihr Argument parallel auswertet und währenddessen das Programm weiter laufen lässt.

`rpar` ist die Strategie, die ihr Argument parallel auswertet und währenddessen das Programm weiter laufen lässt.

`rseq` ist die Strategie, die auf das Ergebnis ihres Argumentes wartet und erst dann mit dem Programm weiter macht.

`rpar` ist die Strategie, die ihr Argument parallel auswertet und währenddessen das Programm weiter laufen lässt.

`rseq` ist die Strategie, die auf das Ergebnis ihres Argumentes wartet und erst dann mit dem Programm weiter macht.

Protips:

`rpar` ist die Strategie, die ihr Argument parallel auswertet und währenddessen das Programm weiter laufen lässt.

`rseq` ist die Strategie, die auf das Ergebnis ihres Argumentes wartet und erst dann mit dem Programm weiter macht.

Protips:

- Ausgewertet wird jeweils zur WHNF (wenn nichts anderes angegeben wurde).

`rpar` ist die Strategie, die ihr Argument parallel auswertet und währenddessen das Programm weiter laufen lässt.

`rseq` ist die Strategie, die auf das Ergebnis ihres Argumentes wartet und erst dann mit dem Programm weiter macht.

Protips:

- Ausgewertet wird jeweils zur WHNF (wenn nichts anderes angegeben wurde).
- Wird `rpar` ein bereits evaluierter Ausdruck übergeben, passiert nichts, weil es keine Arbeit parallel auszuführen gibt.

`rpar` ist die Strategie, die ihr Argument parallel auswertet und währenddessen das Programm weiter laufen lässt.

`rseq` ist die Strategie, die auf das Ergebnis ihres Argumentes wartet und erst dann mit dem Programm weiter macht.

Protips:

- Ausgewertet wird jeweils zur WHNF (wenn nichts anderes angegeben wurde).
- Wird `rpar` ein bereits evaluierter Ausdruck übergeben, passiert nichts, weil es keine Arbeit parallel auszuführen gibt.

Sehen wir uns das mal *in action* an...

Ein Beispiel (1):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

Ein Beispiel (1):

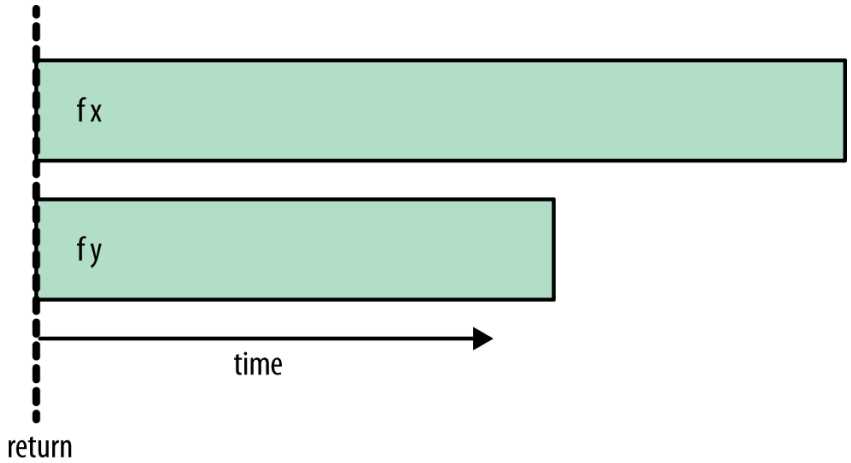
Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

```
-- don't wait for evaluation
runEval $ do
    a <- rpar (f x)
    b <- rpar (f y)
    return (a,b)
```

Ein Beispiel (1):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

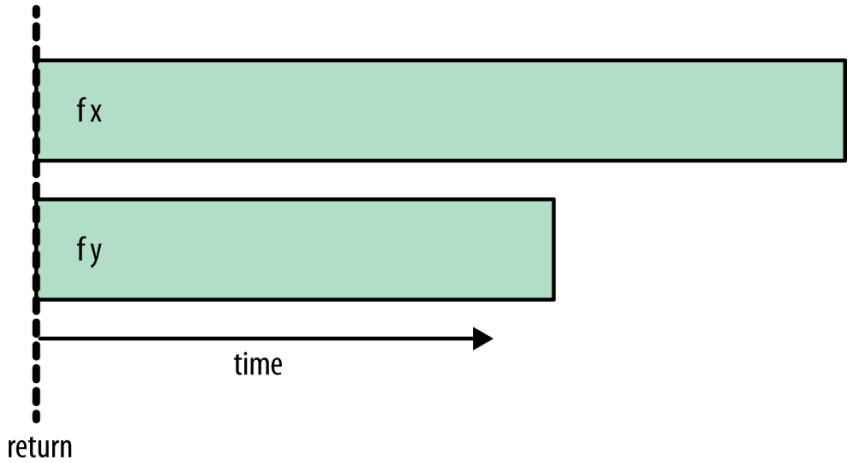
```
-- don't wait for evaluation
runEval $ do
    a <- rpar (f x)
    b <- rpar (f y)
    return (a,b)
```



Ein Beispiel (1):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

```
-- don't wait for evaluation
runEval $ do
    a <- rpar (f x)
    b <- rpar (f y)
    return (a,b)
```



Hier passiert das `return` sofort. Der Rest des Programmes läuft weiter, während `(f x)` und `(f y)` (parallel) ausgewertet werden.

Ein Beispiel (2):

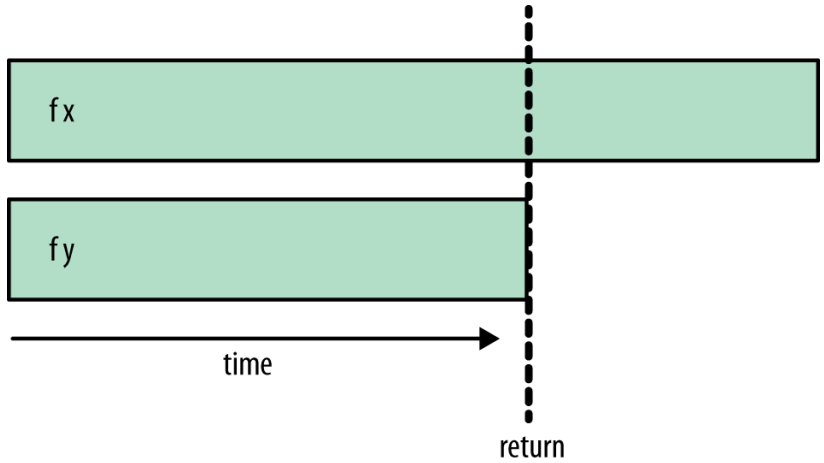
Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

```
-- wait for (f y)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y) -- wait
    return (a,b)
```

Ein Beispiel (2):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

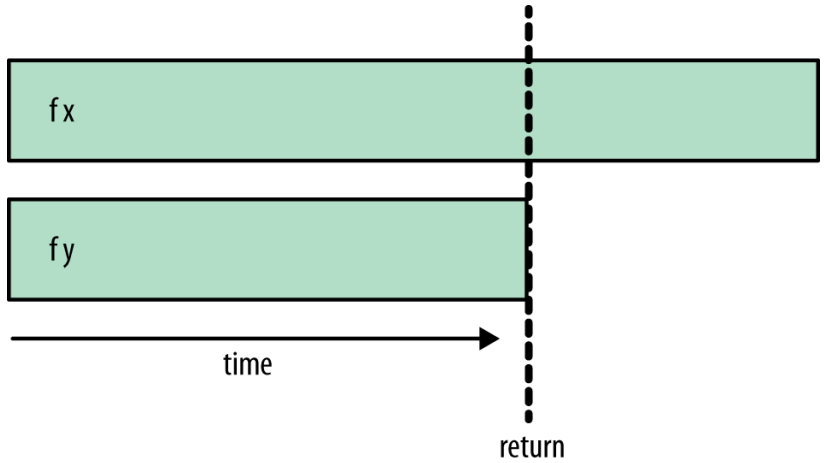
```
-- wait for (f y)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y)  -- wait
    return (a,b)
```



Ein Beispiel (2):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

```
-- wait for (f y)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y)  -- wait
    return (a,b)
```



Hier werden `(f x)` und `(f y)` ebenfalls ausgewertet, allerdings wird mit `return` gewartet, bis `(f y)` zu Ende evaluiert wurde.

Ein Beispiel (3):

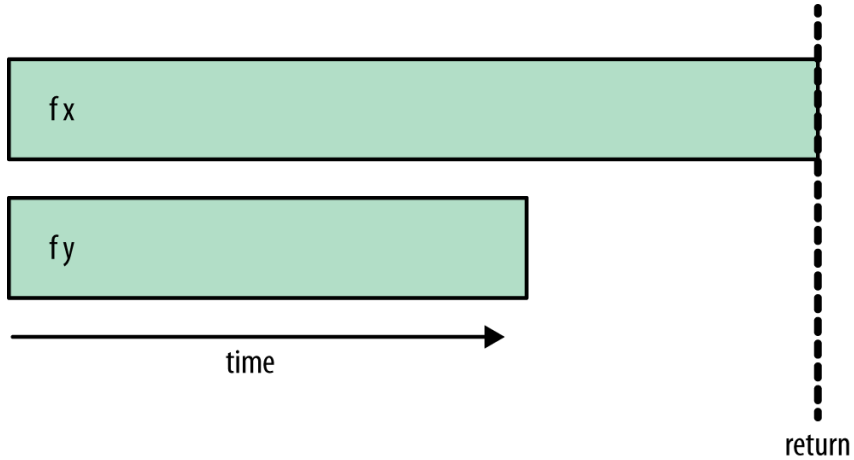
Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

```
-- wait for (f y) and (f x)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y) -- wait
    rseq a          -- wait
    return (a,b)
```

Ein Beispiel (3):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

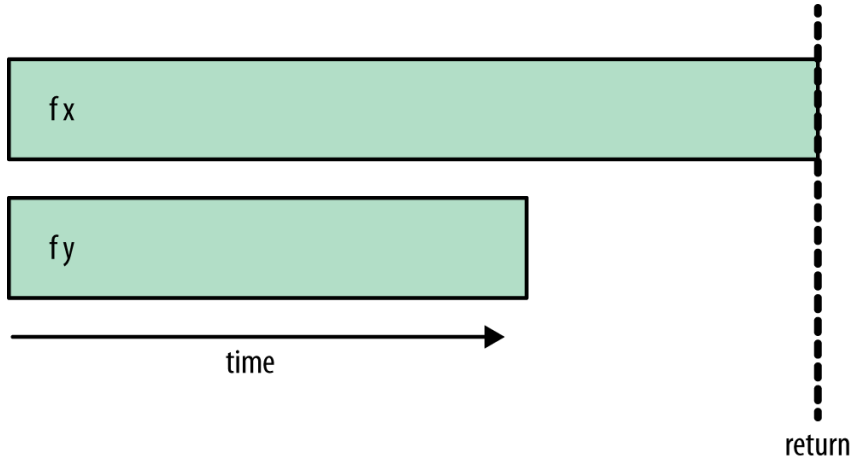
```haskell
-- wait for (f y) and (f x)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y)  -- wait
    rseq a           -- wait
    return (a,b)
```



Ein Beispiel (3):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

```
-- wait for (f y) and (f x)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y)  -- wait
    rseq a           -- wait
    return (a,b)
```

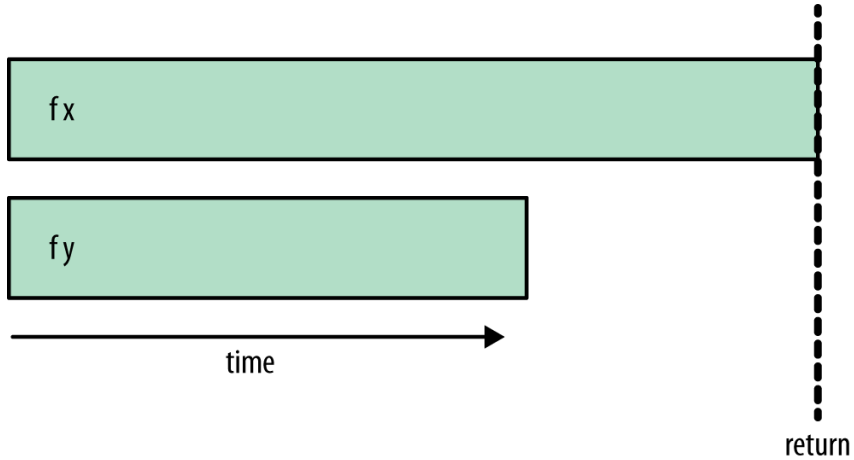


In diesem Code wird sowohl auf `(f x)` als auch auf `(f y)` gewartet, bevor etwas zurück gegeben wird.

Ein Beispiel (3):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

```
-- perhaps more readable:
runEval $ do
    a <- rpar (f x)
    b <- rpar (f y)
    rseq a              -- wait
    rseq b              -- wait
    return (a,b)
```



In diesem Code wird sowohl auf `(f x)` als auch auf `(f y)` gewartet, bevor etwas zurück gegeben wird.

Parallelism

- Die `Eval`-Monade und Strategies
- ○ Die `Par`-Monade
- Die `RePa`-Bibliothek
- GPU-Programming mit `Accelerate`

Parallelism

- Die `Eval`-Monade und Strategies
- Die `Par`-Monade
- ○ Die `RePa`-Bibliothek
- GPU-Programming mit `Accelerate`

Parallelism

- Die `Eval`-Monade und Strategies
- Die `Par`-Monade
- Die `RePa`-Bibliothek
- ○ GPU-Programming mit `Accelerate`