7.6 Nebenläufige funktionale Programmierung

Es gibt viele verschiedene Ansätze zur Erweiterung von funktionalen Programmiersprachen um Nebenläufigkeit. Zunächst betrachten wir einen Ansatz, der ohne eine für den Benutzer sichtbare Spracherweiterung auskommt: Concurrent Haskell.

7.6.1 Concurrent Haskell

Concurrent Haskell (Peyton Jones et al., 1996) erweitert die Sprache Haskell um Möglichkeiten zur nebenläufigen Programmierung. Hierzu wird keine syntaktische Erweiterung vorgenommen, sondern die Konstrukte zur nebenläufigen Programmierung werden durch Bibliotheken bereit gestellt.

Die Basisbibliothek zu Concurrent Haskell ist die Bibliothek `Control.Concurrent`.[1] Diese gehört zu den Standardbibliotheken vom GHC, d.h. sie kann ohne Installationim GHCi geladen werden:

```
> ghci
GHCi, version...
Prelude> :module Control.Concurrent
```

In dieser Bibliothek sind verschiedene Operationen zur nebenläufigen Programmierung in Concurrent Haskell als I/O-Operationen definiert. Dies hat den Effekt, dass die referenzielle Transparenz der Kernsprache erhalten bleibt. Zur Erzeugung von Prozessen bietet Concurrent Haskell die primitive Operation

```
forkIO :: IO () → IO ThreadId
```

an. Der Rückgabewert von `forkIO` ist ein Wert des abstrakten Datentyps

```
data ThreadId
```

der einen Prozess identifiziert. Durch die Abarbeitung von `(forkIO` a`)` wird die Aktion a nebenläufig zur weiteren Abarbeitung der nachfolgenden Aktionen ausgeführt. Wenn z. B. die Operationen `printLoop` ihr Argument beliebig oft ausgibt, dann werden durch

```
do _ <- forkIO (printLoop 'a')
   printLoop 'z'
```

die Zeichen `a` und `z` je nach Scheduling abwechselnd ausgegeben.

Zur Synchronisation und Kommunikation solcher nebenläufiger Berechnungen bietet Concurrent Haskell veränderbare Variablen an, die entweder leer sind oder einen Wert eines bestimmten Typs `a` enthalten können:

```
data MVar a
```

[1] `http://hackage.haskell.org/package/base/docs/Control-Concurrent.html`

Zur Erzeugung und Manipulation werden folgende Operationen bereit gestellt:

```
newMVar      :: a → IO (MVar a)
newEmptyMVar ::      IO (MVar a)
```

Mit diesen Operationen kann man eine volle bzw. leere neue `MVar` erzeugen.

```
takeMVar :: MVar a → IO a
```

Diese Operation liest den Wert einer `MVar` und leert diese. Falls die `MVar` noch keinen Wert hat, blockiert diese Operation den ausführenden Prozess.

```
putMVar :: MVar a → a → IO ()
```

Diese Operation schreibt einen Wert in eine leere `MVar`. Enthält die `MVar` bereits einen Wert, so wird der schreibende Prozess solange suspendiert, bis ein anderer Prozess die `MVar` wieder geleert hat.

Mit Hilfe dieser primitiven Operationen ist es leicht möglich, komplexere Kommunikationsstrukturen aufzubauen. Zum Beispiel kann man eine Synchronisationsstruktur realisieren, bei der die Übergabe eines Wertes synchron passiert, im Gegensatz zur asynchronen Übergabe bei einer `MVar`.

Beispiel 7.17 (Datenstruktur zur synchronen Wertübergabe)**.**

```
data SyncVar a = SyncVar (MVar a)  -- value to be transferred
                         (MVar ()) -- signal for reader presence

newSyncVar :: IO (SyncVar a)
newSyncVar = do  v <- newEmptyMVar
                 s <- newEmptyMVar
                 return (SyncVar v s)

putSyncVar :: SyncVar a → a → IO ()
putSyncVar (SyncVar v s) val = do putMVar  v val
                                  takeMVar s

takeSyncVar :: SyncVar a → IO a
takeSyncVar (SyncVar v s) = do putMVar  s ()
                               takeMVar v
```

In ähnlicher Weise kann man weitere Kommunikationsabstraktionen, wie z. B. Kanäle (Channels) zum Puffern mehrerer Elemente, durch geeignete Verwendung von `MVar`s definieren.

7.6.2 Erlang

Als Beispiel für eine Sprache, bei der Konzepte zur nebenläufigen Programmierung in die Sprache selbst integriert wurden, betrachten wir die Sprache **Erlang** (Armstrong et al., 1996):

- Entwickelt von der Firma Ericsson zur Programmierung von Telekommunikationssystemen
- Anwendungsbereich: komplexe, verteilte Systeme
- Entstanden als Mischung logischer und strikter funktionaler Sprachen, die um ein Prozesskonzept erweitert wurde

Erlang-Programme enthalten folgende Konstrukte:

- Ein Programm ist eine Menge von Modulen, wobei jedes Modul eine Menge von Funktionsdefinitionen enthält.
- Datentypen: Zahlen, Atome, Pids (process identifiers), Tupel, Listen
- Pattern matching bei Funktionsdefinitionen und(!) bei der Kommunikation

Beispiel 7.18 (Listenkonkatenation in Erlang).

```
append([],    Ys) -> Ys;
append([X|Xs],Ys) -> [X|append(Xs,Ys)].
```

Die Syntax ist Prolog-ähnlich: Variablen beginnen mit Großbuchstaben, alles andere mit Kleinbuchstaben. Eine Erweiterung gegenüber Prolog sind Tupel. Z.B. ist `{a,1,b}`, ein Tupel mit den Komponenten `a`, `1` und `b`.

Für die Programmierung mit mehreren Prozessen bietet Erlang drei Konstrukte an.

- Prozess erzeugen

  `spawn(<module>,<function>,<arglist>)`

  Dies erzeugt einen neuen Prozess, der die Berechnung „`<function>(<arglist>)`“ durchführt. Das Ergebnis dieses `spawn`-Aufrufs ist ein Pid (process identifier).

  Wichtig: Jeder Prozess hat eine Nachrichtenwarteschlange (mailbox), die er lesen kann und in die andere schreiben können, indem sie diesem Prozess Nachrichten schicken.

- Nachricht senden

  `<Pid>!<msg>`

Dieses Konstrukt sendet eine Nachricht (einen Wert) `<msg>` an den Prozess `<Pid>`.

- Nachricht empfangen

  ```
  receive
    <msg_1> -> <action_1>;
    ...
    <msg_n> -> <action_n>
  end
  ```

  Hierbei ist `<msg_i>` das *Muster* einer Nachricht, die in der mailbox gesucht wird, und `<action_i>` eine Sequenz von Funktionsaufrufen.

  Die Bedeutung diese Konstrukts ist: Suche in der Nachrichtenwarteschlange (mailbox) die erste Nachricht, die zu einem `<msg_i>` passt (und zu keinem `<msg_j>` mit $j < i$) und führe `<action_i>` aus.

Beispiel 7.19 (Ein einfacher `echo`-Prozess)**.**

```
- module(echo)                % Modulname
- export([start/0, loop/0]).  % exportierte Funktionen

start() -> spawn(echo, loop, []).  % starte echo-Prozess

loop() -> receive
            {From,Msg} -> % empfange Paar von Pid und Inhalt
               From!Msg,  % sende Inhalt zurück
               loop()
          end.
```

Beispiellauf:

```
...
Id = echo:start(),
Id!{self(),hallo},  % self: pid des eigenen Prozesses
...
```

Hierbei kann man eine wichtige Programmiertechnik von Erlang und der nachrichtenbasierten Programmierung sehen. Um eine Nachricht an einen Sender zurückzuschicken, muss der Sender in der Nachricht seinen Pid mitschicken, damit der Empfänger an diesen Antworten kann.

Beispiel 7.20 (Ping-Pong in Erlang)**.** Hier verwenden wir Ping/Pong-Nachrichten statt Tupel wie in Linda:

```
- module(pingTest).          % Modulname
- export([main/1, pong/0]).  % exportierte Funktionen

% n-mal Ping:
ping(Pid,0) -> okay;
ping(Pid,N) -> Pid!{self(),ping},
               receive
                 pong -> ping(Pid,N-1)
               end.

pong() -> receive
            {Pid,ping} -> Pid!pong,
                          pong()
          end.

main(N) -> Pid = spawn(pingTest, pong, []),  % starte pong-Prozess
           ping(Pid,N).
```

Wichtige Aspekte von Erlang sind

- einfache Verarbeitung von Nachrichten durch Pattern matching
- kompakte Definition von Funktionen
- massive Codereduktion (auf 10% – 20% der ursprünglichen Größe)
- Ein **Codeaustausch** ist im laufenden(!) Betrieb möglich: Ersetze dazu im obigen `echo`-Programm den rekursiven Aufruf „`loop()`“ durch „`echo:loop()`“. Dies hat den folgenden Effekt:
  - Benutze im rekursiven Aufruf die aktuelle Version der Funktion `loop()` im Modul `echo`.
  - Falls das Modul `echo` verändert und neu compiliert wird, so wird beim nächsten Aufruf die neue veränderte Version benutzt.
- Der Übergang zu verteilter Programmierung ist einfach: Starte Prozesse auf anderen „Knoten“ (Rechnern), wobei am Code der Kommunikation nichts geändert werden muss.