Erweiterung von Haskell um portbasierte Kommunikation zur verteilten Programmierung

Mathematisch-Naturwissenschaftliche Fakultät

Diplomarbeit im Fach Informatik

Vorgelegt von:

12. Semester Informatik
Matrikelnummer: 190959

ulrich@norbisrath.de

Gutachter:
Professor Dr. K. Indermark
Professor Dr. M. Hanus

Angefertigt am:
LEHRSTUHL FÜR INFORMATIK II
(PROGRAMMIERSPRACHEN UND VERIFIKATION)

bei
Professor Dr. K. Indermark

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel genutzt habe.

Stolberg, den 13.7.2000

Inhaltsverzeichnis

1. Einleitung ... 1
1.1 Aufbau der Arbeit ... 2
1.2 Einführung in Haskell ... 2
1.3 Nomenklatur und verwendete Zeichen ... 7

2. Kommunikationskonzepte in funktionalen Programmiersprachen ... 9
2.1 Erlang ... 9
2.2 Goffin ... 11
2.3 Eden ... 12
2.4 Curry ... 13
2.5 Erlang-Style Distributed Haskell ... 14

3. Concurrent Haskell ... 17
3.1 Threadkontrolle ... 17
forkIO ... 17
killThread ... 17
raiseInThread ... 18
myThreadId ... 18
threadDelay ... 18
yield ... 18
3.2 Kommunikation und Synchronisation zwischen Threads ... 18
MVars ... 18
Semaphoren ... 20
Channels ... 20
3.3 Zusammenfassung ... 22

4. Portbasierte Kommunikation ... 23
4.1 Vergleich der vorgestellten Konzepte ... 23
Verteilung des Multi-Writer-Multi-Reader-Konzeptes von Concurrent Haskell ... 24
Multi-Writer-Single-Reader Konzeptes für verteilte Systeme ... 26
4.2 Der portbasierte Ansatz ... 28

5. Anforderungsdefinition ... 29
5.1 Datenstrukturen ... 31
Port ... 31
PortHost und PortName ... 32
Link ... 32
5.2 Verwaltungsfunktionen ... 32
newPort ... 33
mergePort, <|> ... 33

registerPort 33
lookupPort 34
lookupLocalPort 34
5.3 Zugriffsfunktionen 34
readPort 34
readPortTimed 34
writePort, <!> 34
writePortFast 35
sendToPort, <!!>, sendToPortFast 35
sendToLocalPort, sendToLocalPortFast 35
5.4 Überwachungsfunktionen 35
link 35
unlink 35
5.5 Weitergereichte Schnittstelle 36
forkIO, killThread, raiseInThread, threadDelay, yield, myThreadId, Threadid 36
try 36

6. Implementation 37
6.1 Grundlegende Struktur 37
6.2 Die Datenstruktur Port 38
Internal Ports 39
Mergeports 43
Die komplette Datenstruktur Port 44
6.3 Schematische Darstellung der Kommunikation 44
6.4 Die Module der Bibliothek 49
Globale Datenstrukturen und Hilfsfunktionen (PortGlobals.hs) 49
Externes Postamt - External Post Office (ExtPostOffice*.hs, NSocket.hs) 50
Internes Postamt - Internal Post Office (IntPostOffice.hs) 57
Portschnittstelle (Port.hs) 59
6.5 Sprachspezifische Probleme 64

7. Benutzung des Portkonzeptes 67
7.1 Voraussetzungen 67
7.2 Tutorium oder mein erstes portbasiertes Haskellprogramm 67
interne Kommunikation 67
externe Kommunikation 68
7.3 Allgemeine Anleitung 70

8. Beispiele 71
8.1 Datenbankserver und -client 71
Schnittstelle 71
Server 72
Client 74
8.2 Chatserver und -client 75
Schnittstelle 76
Server 76
Client 77

9. Zusammenfassung und Ausblick 81

Anhang A:
Benutzte Hard- und Software **83**

Anhang B:
Quelltexte der entwickelten Bibliothek **85**

Quelltextverzeichnis **119**

Abbildungsverzeichnis **121**

Quellenverzeichnis **123**

Index **125**

1. Einleitung

Heutige Anwendungen im IT-Bereich beschränken sich immer weniger auf den lokalen Einsatz auf einem Rechner. Datenkommunikation und Vernetzung stehen im Vordergrund. Viele Anwendungen erfordern die Kommunikation mit weit entfernt liegenden Rechnern: z. B. muss ein Geldausgabeautomat mit einem Bankserver kommunizieren oder im Telekommunikationsbereich müssen Verbindungen über große Entfernungen in kürzester Zeit mit der richtigen Stelle aufgebaut werden. Aber auch nahezu alle Anwendungen im Internet benötigen eine solche Art der Kommunikation. Systeme, die aus mehreren miteinander kommunizierenden Anwendungen bestehen, nennt man »verteilte Systeme«.

Zwei Probleme stehen bei verteilten Systemen im Vordergrund. Dies ist zum einen die Lastverteilung. Auf Grund der Größe eines solchen Systems (man denke nur an die eben angesprochene Kommunikation zwischen Geldausgabeautomaten und Bankserver) ist es oft wünschenswert, die große Anzahl von Anfragen auf mehrere Bankserver zu verteilen. Zum Anderen muss aber auch eine sehr robuste Kommunikation gewährleistet werden. D. h.: das System muss selbst in der Lage sein, Fehler bzw. Ausfälle zu erkennen, darauf zu reagieren und die Kommunikation ggf. umzuleiten.

Eine moderne Programmiersprache sollte also Mechanismen bieten, solche Strukturen zu realisieren und adäquat abzubilden.

Wie man sieht, geht es hier weniger um die Parallelisierung von Anwendungen, also um die Beschleunigung der Berechnung durch Verteilung der Berechnung auf mehrere Computer, als um die Abbildung der verteilten Struktur, die vielen solcher Anwendungen innewohnt.

Da Projekte, die mehrere Anwendungen auf mehreren Rechnern miteinander verbinden und als ein System arbeiten lassen sollen, von Natur aus deutlich komplexer und somit auch bei der Programmierung fehleranfälliger sind als nicht verteilte Anwendungen, sind Programmiersprachen gefragt, die einem einfache Methoden der Verifikation und Entwicklung mit geringer Fehleranfälligkeit an die Hand geben. Funktionale Programmiersprachen bieten durch ihr extrem hohes Abstraktionsniveau Ansätze, diese Kriterien zu erfüllen (siehe 1.2).

Leider gibt es nur wenige funktionierende (d.h. praktisch umgesetzte und lauffähige) Ansätze zur Programmierung verteilter Systeme in funktionalen Programmiersprachen. In dieser Arbeit werden im folgenden Kapitel einige dieser Ansätze skizziert.

Eine relativ weit entwickelte funktionale Programmiersprache ist Haskell [Has]. Doch auch in ihr existiert noch kein funktionierendes Konzept, bis auf den direkten Zugriff auf Betriebssystemroutinen, zur verteilten Programmierung.

Das Ziel dieser Arbeit ist es, für Haskell ein solches Konzept zu entwickeln und zu implementieren.

1.1 Aufbau der Arbeit

Die Arbeit lässt sich grob in drei Teile gliedern. Sie beginnt mit einer theoretischen Motivation eines Konzeptes zur verteilten Programmierung, beschreibt anschließend die praktische Umsetzung dieses Konzeptes in Haskell und endet mit einer Anleitung der Benutzung des Konzeptes und zwei Beispielen.

Im ersten Teil werden zu Anfang Konzepte zur verteilten Kommunikation in anderen funktionalen Programmiersprachen betrachtet. Im folgenden Kapitel wird ein Ansatz in Haskell zur prozessinternen verteilten Kommunikation zwischen Subprozessen (Threads) beschrieben. Am Ende des ersten Teils wird ein neues Konzept zur Kommunikation in verteilten Systemen vorgestellt.

Der zweite Teil beginnt mit der Anforderungsdefinition an das in dieser Arbeit verwirklichte Projekt. Als erstes wird dort die Außensicht der Eigenschaften, die das Projekt haben soll beschrieben. Im Anschluss folgt dann die tatsächliche interne Realisierung mit der genauen Erklärung der internen Abläufe und der technischen Umsetzung.

Die Kapitel des dritten Teils geben eine kleine geführte Einleitung in die Benutzung des hier vorgestellten Konzeptes in Haskell, wie eine allgemeine Anleitung und stellen zwei etwas größere Beispiele, die bei der Motivation zitiert werden, ausformuliert vor.

Wer weniger an dem theoretischen Hintergrund interessiert ist, der sollte Kapitel 5 (Anforderungsdefinition), 7 (Benutzung des Portkonzeptes) und 8 (Beispiele) lesen.

Wer sich für das benutzte Konzept interessiert, sollte sich Kapitel 4 (Portbasierte Kommunikation) und vielleicht im Vergleich zu anderen Konzepten das Kapitel 2 (Kommunikationskonzepte in funktionalen Programmiersprachen) und Kapitel 3 (Concurrent Haskell) ansehen.

1.2 Einführung in Haskell

Haskell ist eine sehr junge Programmiersprache. Sie ist nach dem amerikanischen Logiker Haskell B. Curry benannt (die folgenden Daten stammen aus »Online Haskell Kurs«, [RHi96] und »The Haskell 98 Report«, [JHA99]).

1987 wurde das Haskell-Komitee auf der Functional Programming Languages and Computer Architecture (FPCA) gegründet, dem u.a. Paul Hudak, Simon Peyton Jones und Philip Wadler angehören.

1990 wurde die Haskell 1.0 Sprachdefinition festgelegt.

1992 erschien die Haskell 1.2 Sprachdefinition (in den Sigplan Notices)

1995 erschien ein Vorschlag zu Haskell 1.3 (auch auf der FPCA)

Seit Mitte 1997 gibt es den auch heute noch benutzten Sprachstandard Haskell 98 (The Haskell Report 98, [JHA99] und The Haskell Library Report 1.4 [JHB99]).

Haskell ist eine rein funktionale Programmiersprache.

Ursprünglich bilden funktionale Programme Eingabedaten auf Ausgabedaten ab. Diese Beziehung zwischen Ein- und Ausgabedaten beschreibt man mit Hilfe mathematischer Ausdrücke, indem man elementare Ausdrücke für einfache Funktionen zugrunde legt und hieraus mit Operationen, die auf Funktionen definiert sind, komplexere Funktionen darstellt, insbesondere auch Funktionen, die auf Funktionenräumen definiert sind. Das wichtigste Konstruktionsprinzip ist hierbei die Rekursion. Ein Programm besteht aus einer Menge von Ausdrücken, die Funktionen definieren.

Variablen in funktionalen Programmiersprachen sind nicht wie im Bereich der imperativen Programmierung »Behälter«, denen Werte zugewiesen werden, sondern Platzhalter für Werte und Funktionen, mit denen im mathematischen Sinne symbolisch gerechnet wird.

Speicherverwaltung im Sinne der Allokierung und Freigabe entfällt somit. Aus dieser expliziten Speicherverwaltung resultiert ein hohes Programmierniveau. Funktionale Programmiersprachen haben keine Seiteneffekte und sind auf Grund ihrer mathematischen Struktur leichter zu verifizieren und auf Korrektheit zu überprüfen als imperative Programmiersprachen.

Funktionale Programme werden über ihre Eigenschaften, nicht über ihren zeitlichen Ablauf programmiert. Allerdings erfordert eine solche Programmierung auch ein völlig neues Verständnis von Algorithmik und Programmierung. Programme lassen sich nicht in gewohnt imperativer Weise notieren. Eine ausgesprochen große Planungsphase ist erforderlich, bevor man mit der eigentlichen Implementierung beginnen kann.

Haskell ist eine typisierte funktionale Sprache, d.h. jeder Ausdruck der Sprache hat einen Typ. Dieser Typ kann je nach Komplexität ein Grunddatentyp wie `Bool` oder `Int` aber auch ein komplexer Datentyp wie eine Funktion sein.

Um neue Datentypen zu konstruieren, bietet Haskell Konstruktoren. Die Definition eines neuen Datentypen beginnt in Haskell mit dem Schlüsselwort `data`. Zum besseren Verständnis soll die Konstruktion von Datentypen hier an drei kleinen Beispielen betrachtet werden.

Ein Datentyp `Boolean` ließe sich in Haskell folgendermaßen definieren:

```
data Boolean = True | False
```

Der Datentyp ist in diesem Fall `Boolean` und die Konstruktoren `True` und `False`. Es ist zu beachten, dass Datentypen und Konstruktoren in Haskell immer mit einem Großbuchstaben beginnen müssen. Ein Wert vom Datentyp `Boolean` kann also die Werte `True` oder `False` annehmen.

Es gibt in Haskell auch polymorphe Datentypen. Dabei handelt es sich um eine Art Schablone für einen Datentyp. Ein solch polymorpher Datentyp ist z .B. der Typ `Maybe`. Er wird sehr häufig in der Implementation benutzt.

```
data Maybe t = Just t | Nothing
```

`t` ist dabei ein Platzhalter für einen beliebigen anderen Typ. So ist z.B. die folgende Definition möglich:

```
type MaybeInt = Maybe Int
```

In einem Wert vom Typ `MaybeString` kann nun eine ganze Zahl mit einem vorangestellten `Just` oder `Nothing` stehen. Sicherlich ist aufgefallen, dass hier zur Spezifikation eines neuen Datentyps nicht das Schlüsselwort `data` benutzt wurde sondern `type`. Das Schlüsselwort `data` wird nur bei der Definition eines Datentyps mit Konstruktoren eingesetzt. Zur Spezifikation wird `type` benutzt.

Ein Datentyp, der im Verlauf der Arbeit auch eingesetzt wird, ist der Datentyp `Either`. Er wird folgendermaßen konstruiert:

```
data Either t1 t2 = Left t1 | Right t2
```

Dieser neue Datentyp hat den Typ `Either t1 t2` wobei `t1` und `t2` zwei beliebige andere Datentypen sein können. Er besitzt die Konstruktoren `Left` und `Right`. Das Besondere an diesem Datentyp ist, dass er es einem ermöglicht in einem Datentyp zwei beliebige andere Datentypen aufzunehmen. So würde ein Typ der mit

```
type Eithertest = Either Int String
```

definiert wird, sowohl eine ganze Zahl als auch eine Zeichenkette repräsentieren können.

Ein weiterer wichtiger Datentyp in funktionalen Programmiersprachen wie Haskell sind Funktionen. Diese können in Haskell über zwei Mechanismen definiert werden: direkt oder über Pattern Matching.

Für die Definition der Fakultätsfunktion erhält man in Haskell folgende beiden Varianten:

direkt

```
fac :: Int -> Int
fac n = if (n == 0) then 1 else ((fac (n-1)) * n)
```

oder über Pattern Matching

```
fac :: Int -> Int
fac 0 = 1
fac (n + 1) = (fac n) * (n + 1)
```

Haskell benutzt zum Auswerten von Ausdrücken eine call-by-need Auswertungsstrategie. Dadurch wird es möglich, in Haskell mit unendlichen Objekten zu rechnen.

Z. B. ist `test` in

```
ganzeZahlenAb n= n:(ganzeZahlenAb(n+1))
test = head (ganzeZahlenAb 3)
```

ein wohldefinierter Ausdruck. Er liefert das erste Element der unendlichen Liste `(ganzeZahlenAb 3)`, nämlich 3 selbst.

Diese Auswertungsstrategie hat zur Folge, dass nur das berechnet wird, was tatsächlich auch gebraucht wird. Insbesondere heißt das, dass die Auswertung nicht strikt ist. Deshalb ist die Reihenfolge, in der Unterausdrücke eines Ausdrucks ausgewertet werden, nicht klar. Es ist auch nicht klar, ob überhaupt alle Unterausdrücke ausgewertet werden. Dies hängt von dem Kontext ab, in dem der Ausdruck ausgewertet wird und ist deshalb schwer vorherzusagen.

Gerade aber eine Interaktion mit der Außenwelt erfordert häufig eine genau spezifizierte Reihenfolge und auch die tatsächliche Auswertung bestimmter Ein-Ausgabeoperationen.

Diese Schwierigkeit wird in Haskell überwunden, indem eine Ein-Ausgabe (I/O)-Operation als Zustandswandler aufgefasst wird. Eine solche Operation hat somit den Typ:

```
type IO a = World -> (a, World)
```

Das bedeutet ein Wert (eine I/O-Operation) vom Typ `IO t` nimmt als Eingabe den Zustand der Außenwelt und liefert einen modifizierten Zustand der Außenwelt zusammen mit einem Wert vom Typ `t`. Natürlich muss eine solche Zustandstransformation so implementiert werden, dass sie direkt ausgeführt wird.

Einen Wert vom Typ `IO t` nennt man in Haskell eine »action«. Beispiele für solche actions sind:

```
getLine :: IO String
putStrLn :: String -> IO ()
```

`getLine` liest eine Zeile von der Standardeingabe und `putStrLn` gibt eine Zeile auf der Standardausgabe aus.

actions können nur innerhalb einer Funktion ausgeführt werden, die auch von einem Typ IO ist. Man nennt dieses IO auch die IO-Monade.[1]

Die Ausführungsreihenfolge von actions kann in Haskell mit den infix Operatoren `>>` und `>>=` bestimmt werden.

```
>>  :: IO a -> IO b -> IO b
>>= :: IO a -> (a -> IO b) -> IO b
```

Um eine Zeile vom Benutzer zu lesen und sie anschließend zweimal auf dem Bildschirm auszugeben, kann man in Haskell folgendes schreiben:

1 Für die Quelle dieser Beschreibung und weitere Literaturhinweisen zu monadischem I/O siehe Concurrent Haskell ([JGF96]).

```
getLine >>=
\line -> putStrLn line >> putStrLn line
```

Listing 1.1: Kleines I/O-Beispiel

Dabei ist `\line->E` in Haskell die Lambda-Abstraktion mit dem Parameter `line` und dem Ausdruck `E`.

Es gibt für einen solchen Ausdruck in Haskell auch noch eine andere Schreibweise, die ein wenig an imperative Programmiersprachen erinnert:

```
do
  line <- getLine
  putStrLn line
  putStrLn line
```

Listing 1.2: Kleines I/O-Beispiel in do-Notation

Dies ist die Schreibweise, die in dieser Arbeit benutzt wird.

Oft ist es nützlich, eine action zu haben, welche keine Ein-Ausgabeoperation durchführt und direkt ein bestimmtes Ergebnis zurückliefert:

```
return :: a -> IO a
```

So kann eine action, die eine Eingabeaufforderung ausgibt, anschließend eine Zeile einliest und diese dann als Wert zurückliefert, wie folgt in Haskell implementiert werden:

```
prompt :: IO String
prompt =
  do
    putStrLn "Ich bitte um eine Eingabe"
    line <- getLine
    return line
```

Listing 1.3: Kleines I/O-Beispiel mit return

Mit diesen actions ist es möglich, aus anderen actions neue zusammenzusetzen. Aber wie werden überhaupt actions ausgeführt? Bei Verwendung des Haskell-Compilers ghc ([GHC]) gibt es dafür eine ausgezeichnete Funktion `main` vom Typ `IO ()`, die als erstes aufgerufen wird und den Programmablauf steuert. Aus ihr können andere Funktionen der IO-Monade (actions) aufgerufen werden oder auch unmonadische Funktionen, deren Auswertung dann im klassischen Sinne funktionaler Programmiersprachen abläuft.

Fehler, die beim Ausführen einer action, also einer Funktion der IO-Monade, auftreten, werden durch sogenannte »Exceptions« signalisiert. Diese veranlassen in der Regel den Thread[2], in der eine solche Exception aufgerufen wurde, zu terminieren. Man spricht in diesem Fall davon, dass die aufgerufene Funktion eine Exception »ausgelöst« hat. Um eine solche Exception aufzufangen, kann eine Funktion der IO-Monade durch Anwendung der Funktion `try` auf sie überwacht werden:

```
try :: IO a -> IO (Either Exception a)
```

Falls tatsächlich eine Exception auftritt, liefert `try` mit einem vorangestellten `Left` eine der in Listing 1.4 aufgelisteten Exceptions zurück.[3] Ansonsten wird mit vorangestelltem `Right` der reguläre Rückgabewert der mit `try` überwachten Funktion zurückgeliefert.

```
data Exception
  = IOException        IOError          -- IO exceptions (from 'fail')
  | ArithException     ArithException   -- Arithmetic exceptions
  | ErrorCall          String           -- Calls to 'error'
```

2 Für die Definition eines Threads siehe die Nomenklatur (nächstes Kapitel, 1.3).

3 Ursprünglich fängt der Befehl `try` nur `IOExceptions` auf. Die Variante, des `try`-Befehls, die von der in dieser Arbeit entwickelten Bibliothek exportiert wird, kann allerdings auch die anderen Exceptions auffangen, falls sie innerhalb der überwachten Funktion oder ihrer Unterfunktionen auftreten.

```
  | NoMethodError       String          -- A non-existent method was invoked
  | PatternMatchFail    String          -- A pattern match failed
  | NonExhaustiveGuardsString           -- A guard match failed
  | RecSelError         String          -- Selecting a non-existent field
  | RecConError         String          -- Field missing in record construction
  | RecUpdError         String          -- Record doesn't contain updated field
  | AssertionFailed     String          -- Assertions
  | DynException         Dynamic        -- Dynamic exceptions
  | AsyncException       AsyncException -- Externally generated errors
  | NonTermination                      -- Infinite Loop

data ArithException
  = Overflow
  | Underflow
  | LossOfPrecision
  | DivideByZero
  | Denormal

data AsyncException
  = StackOverflow
  | HeapOverflow
  | ThreadKilled
```

Listing 1.4: Exceptions

Wenn man das Beispiel aus Listing 1.3 um eine Möglichkeit zum Abfangen einer Exception (z.B. wenn der Prozess, der Zeichen für die Standardeingabe generiert unvorhergesehen terminiert) beim Lesen von Daten von der Standardeingabe erweitern möchte, kann man folgendes schreiben:

```
prompt :: IO String
prompt =
  do
    putStrLn "Ich bitte um eine Eingabe".
    linetest <- try getLine
    case linetest of
      (Left _) -> -- Es ist eine Exception aufgetreten
        do
          putStrLn "Eine Exception ist aufgetreten. Es werden keine
Daten"
                  ++ "zurückgeliefert."
          return ""
      (Right line) -> -- Es ist eine Exception aufgetreten
        do
          return line
```

Listing 1.5: Kleines I/O-Beispiel mit Überwachung des Einlesens von der Standardeingabe

Um selber eine Exception innerhalb einer Funktion der IO-Monade auszulösen, kann die Funktion `fail` verwendet werden.

```
fail :: String -> a
```

Wie man an dem generischen Rückgabetyp sieht, lässt sich `fail` auch außerhalb der IO-Monade einsetzen. An dieser Stelle reicht es aber aus zu wissen, dass es sich in einer Funktion, die in der IO-Monade liegt, genau wie jede andere action einsetzen lässt. Wenn die Exception, die durch `fail` ausgelöst wird, nicht via `try` abgefangen wird, wird der übergebene String als Fehlermeldung ausgegeben. Die ausgelöste Exception hat den Konstruktor `ErrorCall`.

Zum Erlernen von Haskell eignen sich der Online Haskell Kurs ([RHi96]), das Haskell Tutorial ([HPF99]) und der Haskell Companion ([Ski99]). Die Sprachreferenz findet man in The Haskell 98-Report ([JHA99]), The Haskell Libraries Report ([JHB99]) und den GHC Haskell Libraries ([HLi]). Beim Erlernen von Haskell ist weiterhin der Interpreter hugs ([HUG]) sehr praktisch, da man bei ihm keine langen Übersetzungszeiten hat und kurze Beispiele direkt an der

Kommandozeile eingeben kann. Allerdings kann man diesen nicht für dieses Projekt einsetzen, da dieses Projekt einige spezielle Module des ghcs benutzt, die im hugs nicht vorhanden oder nicht vollkommen implementiert sind.

1.3 Nomenklatur und verwendete Zeichen

Im Bereich der verteilten Programmierung gibt es viele verschiedene auch sich widersprechende Begriffe für denselben Sachverhalt. Deshalb möchte ich hier die in dieser Arbeit benutzte Begrifflichkeit festlegen. Ich halte mich dabei an die selben Begrifflichkeiten wie sie in einem bekannten Standardwerk der Informatik festgelegt sind: Operating System Concepts, [ASG].

i) Rechner

Unter einem Rechner werde ich eine Einheit verstehen, die sich über eine IP-Nummer identifizieren lässt. Dies ist in der Regel ein physikalische Arbeitsplatz-PC mit einer Netzwerkkarte und Anschluss an ein Netzwerk. Problematisch wird diese Definition erst bei der Simulation virtueller Rechner unter einem anderen, wie z. B. mit VMware ([VMw]) möglich. Deswegen wähle ich die Definition über die IP-Nummer.[4] Man könnte auch von einer Netzwerkeinheit sprechen. In vielen verteilten Umgebungen spricht man auch von Knoten. In den Zeichnungen dieser Arbeit wird ein Rechner als ein mit »Rechner *n*« beschriftetes Rechteck dargestellt, wobei *n* eine beliebige natürliche Zahl ist.

ii) Prozess / Prozess-ID

Auf einem Rechner können mehrere Prozesse laufen. Diese werden durch ihre Prozess-ID eindeutig auf dem Rechner identifiziert. Sie laufen je nach zugrunde liegender Hardware echt parallel oder in bestimmten Verfahren (z. B. Round-Robin, vgl. für mögliche Verfahren auch [ASG]) abwechselnd. Ein Prozess in diesem Sinne ist das, was man normalerweise als ein Programm bezeichnet. Man spricht von Heavy-Weight-Processes. Ein Prozess wird in den folgenden Zeichnungen als ein mit »Prozess *n*« beschriftetes Rechteck mit abgerundeten Ecken dargestellt.

iii) Portbasierte Prozesse / Programme

Unter portbasierten Prozessen bzw. Programmen werde ich die Prozesse bzw. Programme verstehen, die die hier entwickelte Bibliothek benutzen.

iv) Threads

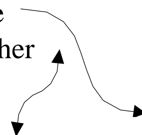
Innerhalb eines Prozesses können mehrere Threads gestartet werden, die auch je nach Hardware echt parallel oder abwechselnd laufen. Diese Art des Laufens nennt man nebenläufig, sowie die Kommunikation zwischen Threads nebenläufige Kommunikation genannt wird. Man nennt bezeichnet Threads auch als Light-Weight-Processes oder auch nur als Prozesse (siehe z. B. Sprachstandard Erlang [AVW96]), weshalb diese Begriffsdefinition hier wichtig ist. Threads werden in den Abbildungen dieser Arbeit als mit »Thread *n*« beschriftete Kreise dargestellt.

v) Kommunikationsverbindungen/ Reader/ Writer

Gerichtete Verbindungen, über die Daten gesendet werden, werden durch Pfeile gekennzeichnet. Dabei ist der Pfeilanfang der Writer der Daten und die Pfeilspitze der Reader der Daten. Eine ungerichtete Verbindung wird durch einen Doppelpfeil gekennzeichnet.

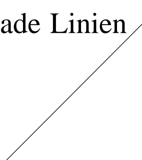
4 Es ist natürlich möglich, dass ein Rechner mehrere IP-Nummern hat. Dies ist prinzipiell aber unbedeutend, da man diese Rechnereinhait auch mit genau einer IP-Nummer eindeutig spezifizieren kann.

Kommunikation zwischen Threads eines Prozesses wird nebenläufige Kommunikation genannt, Kommunikation zwischen Threads unterschiedlicher Prozesse verteilte Kommunikation.



vi) Referenzen

Hält ein Thread/ Prozess Referenzen auf ein Objekt, so werden diese durch gerade Linien gekennzeichnet.



vii) Queues/ Channels

Objekte wie Queues oder Channels, die Daten aufnehmen und wieder abgeben können, werden durch flache Rechtecke dargestellt.

Threads, die in Prozessen auf mehreren Rechnern laufen und miteinander kommunizieren, sehen dann z. B. so aus ():

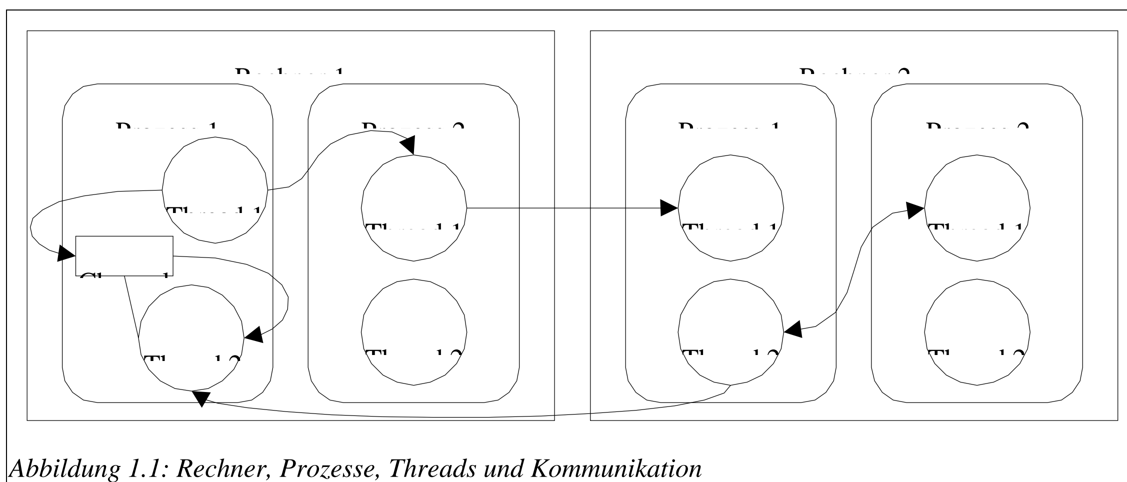


Abbildung 1.1: Rechner, Prozesse, Threads und Kommunikation

Hier schreibt z. B. Thread 2 aus Prozess 1 auf Rechner 2 Daten auf Thread 2 aus Prozess 1 auf Rechner 1 und Thread 2 aus Prozess 1 auf Rechner 1 hält eine Referenz auf einen Channel von Thread 1 und liest von ihm Daten.

2. Kommunikationskonzepte in funktionalen Programmiersprachen

Dieses Kapitel skizziert kurz die Möglichkeiten und Konzepte zur Kommunikation in verteilten Systemen in einigen gängigen funktionalen Programmiersprachen. Ich betrachte hier die Sprachen Erlang, Goffin, Eden, Curry und Erlang-Style Distributed Haskell.

2.1 Erlang

Erlang ist eine moderne funktionale Programmiersprache. Ihr Name stammt von dem dänischen Mathematiker Agner Karup Erlang (1878-1929)[5]. Sie wurde von Ericsson speziell zur Programmierung verteilter Systeme entwickelt. Die Reduktionsstrategie in Erlang ist call-by-value.

Erlang ist ungetypt. Dies bietet Vorteile in der Einfachheit der Kommunikationsmöglichkeit von Threads und Prozessen untereinander, da es nur eine Art von Nachricht gibt, die verschickt wird, allerdings entstehen Nachteile in der Effizienz und der Typsicherheit. Die Aufgabe der Typüberprüfung von Daten wird durch dieses Konzept dem Programmierer übertragen.

Erlang bietet umfangreiche Methoden zur nebenläufigen und verteilten Programmierung.

Im Sprachgebrauch von Erlang ([AVW96] und [Huc99]) können auf einem Rechner mehrere Nodes laufen und in Nodes mehrere Prozesse. Um mit dem in dieser Arbeit (siehe Nomenklatur 1.3) verwendeten Sprachgebrauch konform zu sein, wird hier wieder von Prozessen gesprochen, in denen nebenläufige Threads laufen.

Diese Threads können miteinander mittels asynchronem Message-Passing kommunizieren. Jeder Thread hat eine eindeutige Thread-ID (kurz *pid*[6]). Nachrichten können in Erlang an einen bestimmten Thread durch Angabe seiner *pid* gerichtet werden. Jeder Thread besitzt eine Mailbox, in der alle Nachrichten gespeichert werden, die an ihn gesendet werden. Um die Daten `data` an die *pid* `p` zu senden, schreibt man:

```
p!data
```

Auf die Elemente der Mailbox kann der Thread mittels Pattern-Matching zugreifen.

Mit dem Ausdruck

```
receive
  Message1 -> e1;
  Message2 -> e2;
  ...
  Messagen -> en
end
```

5 Genauere Informationen zum verteilten Programmieren in Erlang als in diesem Kapitel findet man in [AVW96].

6 *pid* ist die Abkürzung von Process-ID, da im ursprünglichen Sprachgebrauch von Erlang von Prozessen, die auf Nodes laufen, die Rede ist. In dem hier verwendeten Sprachgebrauch sind es aber keine Prozesse sondern Threads.

versucht der Thread Nachrichten zu empfangen, die auf eine der Nachrichten $Message_i$ mit i=1,...,n passen und mit dem Ausdruck e_i zu verarbeiten. Dabei werden ungebundene Variablen in $Message_i$ gebunden und können in dem Ausdruck e_i benutzt werden. Bei einer neu eingegangenen Nachricht wird versucht, diese Schrittweise auf eines der Patterns $Message_1$ bis $Message_n$ zu matchen. Bei dem ersten passenden Pattern wird der zugehörige rechts stehende Ausdruck berechnet und die zugehörige Nachricht wird aus der Mailbox entfernt. Passt die Nachricht auf keins der Pattern, so wird die nächste Nachricht aus der Mailbox in dem selben Verfahren bearbeitet. Die nicht passende Nachricht verbleibt in der Mailbox und wird beim nächsten Aufruf von `receive` wieder mit den dann übergebenen Pattern verglichen. Falls keine Nachricht anliegt, suspendiert receive, bis eine Nachricht empfangen wird.

Sind Nachrichten in einem falschen Format (z.B. `sendto` anstatt `sendTo`) verschickt worden verbleiben diese also in der Mailbox und werden nicht erkannt. So können Tippfehler zu Deadlocks führen.

Neue Threads eines Prozesses können unter Erlang mit folgendem Ausdruck gestartet werden:

spawn (module,func,[p_1,...,p_n])

Dabei ist `func` eine exportierte Funktion aus dem Modul **`module`**, die mit den ausgewerteten Parametern p_1,...,p_n als eigenständiger Thread gestartet wird. Das Ergebnis dieses Ausdrucks ist die *pid* des neu gestarteten Prozesses. Um die eigene *pid* zu bestimmen gibt es den Befehl `self()`, der diese zurückliefert.

Da nur der Thread selbst von seiner Mailbox lesen kann und es keine Möglichkeiten gibt, Threads zu starten, die auch lesenden Zugriff auf diese Mailbox haben, gibt es somit auch nur einen Leser dieser Mailbox. Dies hat weitreichende Konsequenzen zugunsten der Robustheit, wie sich im Kapitel 4 zeigen wird.

Für die verteilte Kommunikation benutzt Erlang die gleichen Mechanismen wie für die nebenläufige. Um unterschiedliche Prozesse auf einem Rechner auseinanderzuhalten, hat jeder Prozess eines Rechners einen auf diesem Rechner eindeutigen Namen. Dieser wird dem Prozess beim Start zugewiesen. Um Threads in anderen Prozessen auf demselben oder anderen Rechnern zu starten, gibt es in Erlang eine erweiterte Version des spawn-Befehls:

spawn (process**,module,**func,[p1,...,pn])

Dabei sind alle Parameter außer `process` wie oben. Dieser setzt sich aus dem Namen des Zielprozesses gefolgt von einem @ und dem Namen des Zielrechners zusammen. Auch dieser Befehl liefert die *pid* des soeben gestarteten Threads zurück. Mit diesem Befehl ist es möglich Quelltext auf einem anderen Rechner ausführen zu lassen.

Dieser Befehl stellt zwar ein mächtiges Konzept für die verteilte Programmierung zur Verfügung, doch ist es oft nötig, dass nicht alle Threads hierarchisch von einer Instanz aus gestartet werden, sondern unabhängig voneinander auf anderen Rechnern.

Um nun an den *pid* eines Threads zu kommen, der unabhängig auf einem anderen Rechner gestartet wurde, muss dieser Thread auf dem Prozess, auf dem er läuft, global registriert werden. Dies geschieht mit dem Ausdruck `register(name,p)`. Dadurch wird die *pid* `p` auf dem lokalen Prozess unter `name` registriert.

Um Daten `data` an eine auf dem Prozess `process` unter dem Namen `name` registrierte *pid* zu senden, schreibt man:

```
{process,name}!data
```

Normalerweise wird diese Art der Kommunikation nur zum Initiieren der Verbindung benutzt. Anschließend werden meist die *pid*s ausgetauscht und die restliche Kommunikation über diese abgewickelt.

Mit einem Link-Mechanismus ist es möglich zwei Threads zu verbinden und in dem Fall, wenn ein Thread terminiert (auch unvorhergesehen), automatisch den mit ihm verbundenen darauf reagieren zu lassen. Um einen Link zu einem anderen Thread aufzubauen, muss `link` mit der *pid* des Threads ausgewertet werden, zu dem eine Verbindung hergestellt werden soll:

```
link (Pid)
```

Terminiert nun einer der beiden Prozesse, wird im Standardfall auch der andere terminiert. Ist dieser auch mit anderen Prozessen verbunden, so werden auch diese terminiert. Es ist aber möglich diese Terminierung abzufangen und den betroffenen Thread eine andere Aktion ausführen zu lassen.

Um einen Link wieder abzubauen, muss einer der beiden verbundenen Threads `unlink` mit der *pid* des anderen auswerten:

```
unlink (Pid)
```

Erlang stellt ein sicheres und flexibles Konzept zur Kommunikation in und zur Erstellung von verteilten Systemen zur Verfügung. Einzig nachteilig kann ein nicht vorhandenes Typsystem angesehen werden.

Es wurden zwar Versuche durchgeführt, ein solches zu etablieren (siehe [MWa97] und [AAr98]). Jedoch ist die Kommunikation in beiden Ansätzen weiterhin ungetypt. Beide Ansätze hatten nicht den nötigen Erfolg, eine Bedeutung zu erlangen. Ungetyptes Erlang ist immer noch der Standard.

2.2 Goffin

Goffin ist eine Erweiterung der Sprache Haskell (1.2) und erbt somit alle Eigenschaften dieser Sprache.[7] Ursprünglich war Goffin konzipiert worden, parallele Berechnungen unter einer funktionalen Programmiersprache zu ermöglichen. In einem neuen Paper mit Namen »Distributed Haskell - Goffin on the Internet« ([CGK98]) wird aber eine Erweiterung von Goffin vorgestellt, die verteiltes Programmieren ermöglichen soll. Goffin fügt Haskell ein Konzept zur nebenläufigen constraint-Programmierung und ein spezielles Port-Konzept zur internen und externen Kommunikation hinzu.

Es stellt Mechanismen zum Starten nebenläufiger Threads zur Verfügung. Threads werden in Goffin Agents genannt. Um eine Berechnung von zwei Threads (Agents) `a1` und `a2` durchführen zu lassen, gibt es die Möglichkeit, diese Threads nebenläufig via `a1 & a2` oder sequentiell (also hintereinander) mit `a1 ==> a2` ablaufen zu lassen.

Agents sind in Goffin nicht vom Typ IO () sondern vom Typ O. Sie laufen also nicht in der IO-Monade. Zur Kommunikation zwischen Threads werden sogenannte equational constraints benutzt. Mit dem Ausdruck `ex x in x:=:f a & g x` können der Agent `f` und `g` über die freie Variable `x` kommunizieren. Der Benutzer muss somit den Umgang mit concurrent constraint programming lernen.

Goffin bietet ein Verfahren, um mehrere Datenströme in einen Datenstrom zu verschmelzen (indeterminate choice):

```
merge :: [a] -> [a] -> [a] -> O
merge xs ys zs =
  choice
    {[]<-xs,[]<-ys} -> zs :=:[]
    {(x:xs')<-xs} -> ex zs' in zs:=:(x:zs')==> merge xs' ys zs'
    {(y:ys')<-ys} -> ex zs' in zs:=:(y:zs')==> merge xs ys' zs'
```

Dieses merge funktioniert nicht für unterschiedliche Typen.

7 Die Informationen zu diesem Kapitel wurden [CGK98] und [Huc99] entnommen.

Um gewisse zeitliche Grenzen einzuhalten, stellt Goffin temporal constraints mit dem `after`-Konstrukt zur Verfügung. Damit wird es möglich, bei Schreib- oder Lesezugriffen auf Variablen nicht beliebig lange zu suspendieren.

Für die Kommunikation mit Threads in externen Prozessen gibt es port-constraints. Mit ihnen sind intern und extern *n* zu *m* Kommunikationen möglich. Es kann also mehrere Leser und mehrere Writer geben, die auf einen Port suspendieren. Um auf Ports zu schreiben und von ihnen zu lesen gibt es folgende Konstrukte:

```
(<<-) :: [a] -> Port a -> O       -- pull message out of a port
(->>) :: [a] -> Port a -> O       -- inject message into a port
```

Um mit unabhängig gestarteten Prozessen über Ports zu kommunizieren, gibt es die Möglichkeit, diesen einen auf dem Rechner eindeutigen Namen zuzuweisen und diesen von anderen Rechnern aus wieder nachzusehen. Allerdings muss für die externe Kommunikation eine textuelle Darstellung von Werten des Typs `a` existieren, die sich auch wieder in Werte vom Typ `a` zurückverwandeln lässt (der Typ `a` muss in Haskell den Klassen Show und Read angehören).

```
named :: (Show a,Read a) => Port a -> String -> O
access :: (Show a,Read a) => String -> Port a -> O
```

Als String übergibt man wie bei Erlang (siehe 2.1) den Namen des gesuchten Port gefolgt von einem `@` und dem Zielrechner.

Auch Goffin beinhaltet viele mächtige Mechanismen zur nebenläufigen und verteilten Programmierung. Allerdings ist die Denkweise mit constraints zu programmieren nicht sehr einfach zu erlernen. Ein Nachteil ist auch, dass die Funktionen zur Kommunikation nicht in die IO-Monade eingebettet sind und sich so nicht mit anderen Betriebssystemfunktionen serialisieren lassen. Der größte Nachteil ist aber sicher, dass es noch keine funktionierende Implementierung von Goffin gibt.

2.3 Eden

Auch Eden ist eine Erweiterung von Haskell (1.2), die aber speziell für paralleles Programmieren entwickelt wurde. Eden unterscheidet nicht zwischen Prozessen und Threads, weshalb in diesem Kapitel nur der Begriff Prozess benutzt wird. Alle Prozesse werden hierarchisch von einem Startprozess aus gestartet und ggf. automatisch verteilt. Dynamisch hergestellte Verbindungen sind nicht im Konzept vorgesehen.

Threads werden in Haskell mit einer festen Anzahl Ein- und Ausgabeströme gestartet. Eine Prozessabstraktion ist im Sinne Edens eine Abbildung von Eingabeströmen `in1`, ..., `inm` in Ausgabeströme `out1`, ..., `outn`:

```
p :: t1 -> ... -> tk -> Process (t1', ..., tm') (t1'', ..., tn'')
p par1 ... park = process (in1, ..., inm) -> (out1, ..., outn)
                  where equation1 ... equationr
```

Dabei stehen in den `equations` die Bestimmungen der Ausgabeströme in Abhängigkeit der Ströme und Parameter `par1` bis `park`.

Der Prozess wird in dem Moment gestartet, wenn eine Prozessabstraktion auf ein Tupel von inport (Eingabestrom) -Ausdrücken angewendet wird. Dies nennt man in Eden eine Prozessinstanziierung. Diese definiert das Tupel der outports (Ausgabeströme) des neu angelegten Prozesses.

Eine Prozessinstanziierung sieht so aus:

```
(out1, ..., outn) = (p e1 ... ek)#(in_exp1,...,in_expm)
```

Die linke Seite ist in diesem Fall das Tupel der Ausgabeströme des erstellten Prozesses.

Auch Eden bietet einen Mechanismus, um auf mehrere Eingabeströme zu suspendieren, diese also in einen Strom zu verschmelzen.

Allerdings ist es auch möglich, einen Ausgangstrom in mehrere Eingabeströme unterschiedlicher Prozesse fließen zu lassen, womit mehrere Reader von einem Writer bedient werden können.

Da in Eden Prozesse nur hierarchisch von einem Prozess aus starten können, ist es für verteilte Systeme, die Prozesse unabhängig voneinander starten müssen ungeeignet.

Die Informationen zu Eden stammen aus [BLO96] und [Huc99].

2.4 Curry

Curry ist eine multi-paradigmatische deklarative Programmiersprache aus funktionalen, logischen und - für uns besonders interessant - nebenläufigen Programmierparadigmen[8]. Aufgrund dieser Kombination von Programmierparadigmen setzt Curry auch eine Kombination unterschiedlicher Auswertungstrategien ein, wie Reduktion, nichtdeterministische Suche von Lösungen und Residuation und Narrowing. Für eine genaue Beschreibung des Berechnungsmodells siehe »A Unified Computation Model for Declarative Programming«, [Han97].

Datentypen werden in Curry genau wie in Haskell (1.2) angegeben. Terme, die aus Konstanten und den Konstruktoren dieser Datentypen zusammengesetzt sind, nennt man in Curry Datenterme. Funktionsapplikationen werden ebenfalls wie in Haskell notiert. Auch das I/O-Konzept mit der Kapselung der I/O-Operationen in einer IO-Monade übernimmt Curry von Haskell. Funktionen sind in Curry Operationen auf Datentermen, deren Bedeutung durch sogenannte *conditional rules* der allgemeinen Form `l | c = r where vs free` angegeben wird. Dabei hat `l` die Form `f t1 ... tn` wobei `f` eine Funktion ist und die `t1`, ..., `tn` Datenterme sind. Die Bedingung `c` ist eine constraint. `r` ist ein wohlgeformter Ausdruck, in dem auch Funktionen, Funktionsaufrufe und freie Variablen vorkommen dürfen und `vs` ist die Liste der freien Variablen, die in `c` und `r`, aber nicht in `l` vorkommen. Eine conditional rule kann angewendet werden, wenn die constraint (`c`) erfüllbar ist. Ein Curry-Programm ist eine Menge von Datentypdeklarationen und conditional rules.

Das Starten nebenläufiger Berechnungen entspricht den Mechanismen zum Starten nebenläufiger Berechnungen in Goffin (2.2). Um zwei Funktionen `a1` und `a2` mit dem Ergebnistyp constraint nebenläufig als Thread auszuführen, schreibt man in Curry `a1 & a2`. Um sie nacheinander auszuführen, schreibt man `a1 &> a2`.

Curry bietet genau wie Goffin equational, port und temporal constraints:

i) equational constraints

Diese sind Gleichungen der Form `e1 =:= e2`.

Eine solche constraint ist genau dann erfüllt, wenn `e1` und `e2` zu dem gleichen Datenterm reduziert werden können.

Auch mit diesen constraints ist bereits eine Kommunikation in nebenläufigen Systemen möglich. Diese ist aber teilweise etwas unhandlich.

ii) port constraints

Zur Vereinfachung der nebenläufigen und für die verteilte Programmierung werden port constraints eingeführt. Um einen Port zu kreieren, muss die constraint `openPort p s` ausgewertet werden, wobei `p` und `s` uninstantiierte freie Variablen sein müssen. p identifiziert man dabei mit dem Port, auf den Daten gesendet werden, und `s` mit dem Strom der

8 Die Informationen zu Curry sind dem Paper [Han99] entnommen.

eingehenden Daten. Es handelt sich hier also um zwei unterschiedliche Strukturen. Die Funktion `openPort` hat folgende Signatur:

```
openPort :: Port a -> [a] -> Constraint
```

Mit der Auswertung von `send m p` wird `m` an den Port `p` gesendet. Um von dem Port zu lesen, muss einfach auf Daten aus dem Eingabestrom s zugegriffen werden.

Um externe Kommunikation zu ermöglichen, werden zwei Funktionen eingeführt, die in der IO-Monade liegen. Mit `openNamedPort n` kann ein Strom eingehender Daten erzeugt werden, auf den von außen Daten gesendet werden können. Wird `openNamedPort` auf dem Rechner *`Host`* ausgeführt, und ist `n` gleich *`Name`*, so erhält man mit `connectPort "`*`Host@Name`*`"` einen Port, auf den man Daten schreiben kann.

Da der Strom eingehender Daten über einen anderen Strom verschickt werden kann, sind auch in Curry, anders als bei Erlang (2.1), mehrere Leser bei mehreren Writern möglich.

iii) temporal constraints

Mit der temporal constraint `after t` kann garantiert werden, dass auf eine ungebundene Variable nicht länger als t Millisekunden suspendiert wird.

Die Verbindung von funktionalen, logischen und nebenläufigen Programmierparadigmen birgt enorme Möglichkeiten. Es lassen sich hier sogar Funktionen zur Berechnung auf anderen Servern, sogenannten Berechnungsservern, über das Netz verschicken.

Im Gegensatz zu Goffin ist die Umsetzung von Curry bereits sehr weit fortgeschritten. Probleme treten aber noch bei Multiple Reader - Multiple Writer Zugriffen auf, wenn einzelne verteilte Prozesse in einem solchen System unvorhergesehen terminieren.

2.5 Erlang-Style Distributed Haskell

Diese Erweiterung von Haskell beruht auf der Idee, das Konzept zur verteilten Kommunikation von Erlang (2.1) in Haskell zur Verfügung zu stellen.[9]

Alle in Erlang vorgestellten Mechanismen werden in diesem Projekt unter Haskell zur Verfügung gestellt.

Jedem Thread wird eine im Prozess eindeutige *pid* und eine Mailbox zugeordnet.

Aus diesen Mailboxen lassen sich Nachrichten mittels Pattern-Matching auswählen. Diese werden bei einem Match entfernt und die zugehörige Funktion wird ausgewertet.

Der Hauptnachteil von Erlang ist, dass es ungetypt ist. Nachrichten, die im falschen Format (mit falschem Typ) gesendet werden, können zu einem Deadlock führen. Dies wird in dieser Erweiterung vermieden. Es tritt ein Laufzeitfehler auf. Dadurch wird es möglich, auf einen solchen Fehler zu reagieren.

Da Haskell noch kein ausgereiftes Laufzeittypsystem besitzt[10], müssen alle Nachrichten, die ein Thread verstehen und an andere Threads versenden können soll, in einem Typen, der zu diesem Thread gehört, in der Thread-Definition mit angegeben werden. Eine Thread[11]-Definition in

9 Die Idee stammt von Frank Huch. Das Projekt wird momentan in einer Diplomarbeit von Volker Stolz umgesetzt. Nähere Informationen lassen sich der Quelle [Huc99] und ggf. später der eben erwähnten Diplomarbeit entnehmen.

10 Im ghc und hugs gibt es Ansätze für ein solches Laufzeittypsystem (siehe Typable und Dynamic in [HLi]). Leider sind aber nur wenig Typen in der Klasse Typable, so dass dort noch ein bisschen Entwicklungszeit benötigt wird.

11 Hier wird das Schlüsselwort `process` anstatt `thread` benutzt, da sich der Sprachgebrauch von [Huc99] von dem in dieser Arbeit verwendeten unterscheidet.

Erlang-Style-Distributed Haskell beginnt analog zu Modul-Definitionen in Haskell mit folgender Definition:

```
process procName (functions to start the process) where
```

Die Funktionen, mit denen der Thread gestartet werden kann (`functions to start the process`), müssen den Typ `IO ()` haben.

Dieser Definition sollte eine Definition folgen, die die Nachrichten definiert, die der Thread versteht und versenden kann:

messages $C_1\ t_{11}\ \ldots\ t_{n1}\ |\ \ldots\ |\ C_m\ t_{m1}\ \ldots\ t_{mn_m}$

Dabei sind die C_i Konstruktoren vom Typ t_{i1} `->` ...`->` t_{in_i} `->` `Msg` mit `Msg` ist der Typ der Nachricht eines Threads. Damit diese Nachrichten auch über das Netz geschickt werden können, müssen alle Typen t_{ij} in den Klassen `Show` und `Read` liegen. Es müssen also textuelle Darstellungen der Typen existieren, die sich wieder in den Typen zurückverwandeln lassen.

Neue Threads werden mit dem Befehl `spawn` gestartet:

```
spawn :: IO () -> IO Pid
```

`spawn` liefert die *pid* des soeben gestarteten Threads zurück. Die Nachrichten in Erlang-Style Distributed Haskell werden an diese *pid*s gesandt.

Dies geschieht mir der Hilfe von `send` oder `<!>`:

```
send, (<!>) :: Pid -> a -> IO a
```

Um Daten zu empfangen gibt es analog zu Erlang eine `receive`-Anweisung, die wie Haskells `case`-Anweisung aufgebaut ist.

```
receive
  p1 | g1 -> e1
       .
       .
       .
  pn | gn -> en
```

Diese Anweisung blockiert den aufrufenden Thread bis eine Nachricht empfangen wurde, die auf eines der Pattern p_i matcht. Das erste passende Pattern wird ausgewählt, die Nachricht aus der Mailbox gelöscht und die entsprechende rechte Seite wird ausgeführt. Aus praktischen Gründen und insbesondere in Hinblick auf spätere Verteilung ist Kommunikation in Erlang-Style Distributed Haskell strikt.

Auch die Funktion `self` ist in dieser Erweiterung abgebildet:

```
self :: IO ()
```

Um verteilt zu kommunizieren, steht eine globale Registrierfunktion zur Verfügung, die einem Thread eines Prozesses einen auf dem Rechner eindeutigen Namen zuweist. Der Prozess selbst ist mit in der *pid* kodiert und das Senden von Daten funktioniert genauso wie bei nebenläufigen Systemen mit send oder <!>.

Zum globalen Registrieren ist eine Anweisung folgendes Typs zu benutzen:

```
register :: String -> IO ()
```

Zum Senden kann man eine der beiden folgenden Varianten benutzen:

```
remoteSend :: Hostname -> String -> a -> IO a
(<!!>) :: (Hostname,String) -> a -> IO a
```

Andere Threads auf anderen Rechnern zu starten ist vorerst in diesem Konzept nicht vorgesehen.

Bis auf die letzte Einschränkung verhält sich ein solches System genauso wie ein entsprechendes Erlang-System. Man erhält zusätzlich noch eine gewisse Sicherheit durch Haskells Typsystem und vermeidet Deadlocks durch falsch formatierte Daten.

3. Concurrent Haskell

Zur Programmierung nebenläufiger Systeme existiert bereits ein Konzept in Haskell nämlich Concurrent Haskell ([JGF96]). Da dieses eine wesentliche Grundlage der praktischen Umsetzung dieser Arbeit darstellt, sollen die zur Verfügung gestellten Funktionen und Datentypen an dieser Stelle etwas ausführlicher besprochen werden als die Funktionen der Sprachen im letzten Kapitel.

In den Bibliotheken des ghc ([GHC], [HLi]) ist Concurrent Haskell bereits vollständig umgesetzt, in hugs ([HUG]) nur zum Teil. Da in dieser Arbeit die volle Funktionalität benötigt wird, werde ich im weiteren nur noch auf die Umsetzung der Mechanismen im ghc eingehen. Es bietet Methoden, um nebenläufige Threads zu starten und zu synchronisieren. Es bietet shared Variables und ein Channelkonzept zur asynchronen Kommunikation. In Concurrent Haskell sind keine Konzepte zur verteilten Kommunikation enthalten. Die Entwicklung und Umsetzung eines solchen Konzeptes ist schließlich Aufgabe dieser Arbeit.

In diesem Kapitel werden die Möglichkeiten zur Threadkontrolle und Threadkommunikation in Concurrent Haskell vorgestellt.

3.1 Threadkontrolle

Um Threads zu starten und wieder zu terminieren und ihr Laufzeitverhalten zu beeinflussen bietet Concurrent Haskell die Funktionen `forkIO`, `killThread`, `raiseInThread`, `myThreadId`, `threadDelay`, und `yield`.

a) forkIO

Um einen neuen Thread anzulegen benutzt man in Concurrent Haskell `forkIO`.

```
forkIO :: IO () -> IO ThreadId
```

Um eine Funktion `f` der IO-Monade mit Parametern `pi` mit `i`=1,2,...,n als Thread zu starten, schreibt man:

```
forkIO (f p1 p2 ... pn)
```

forkIO lieferte einen Wert vom Typ `ThreadId` zurück. Dies ist ein abstrakter Typ, der eine Referenz auf den neu erstellten Thread repräsentiert. Man spricht deshalb auch von einer Thread-ID. Durch Übergabe dieser ID an `killThread` kann man den mit `forkIO` gestarteten Thread von einem anderen Thread aus beenden.

b) killThread

```
killThread :: ThreadId -> IO ()
```

Diese Funktion terminiert den Thread mit der übergeben `ThreadId`.

Der Speicher, der von dem Thread belegt wurde, wird zur Garbagecollection freigegeben, wenn er nicht noch von einer anderen Stelle referenziert wird.

Etwas allgemeiner als `killThread` ist `raiseInThread`.

c) raiseInThread

```
raiseInThread :: ThreadId -> Exception -> IO ()
```

Diese Funktion löst in dem Thread mit der übergebenen Thread-ID eine beliebige Exception aus, die in `Exception` übergeben werden kann.

Tatsächlich löst `killThread` nur die `ThreadKilled` Exception in dem Zielthread aus. Der Zielthread wird gestoppt, auch wenn er auf einen Port suspendiert, und muss die Exception abarbeiten. Wird die Exception nicht abgefangen, so wird der Thread beendet.

Eine wichtige Eigenschaft des `raiseInThread`- (und deshalb auch `killThread`-) Aufrufs ist, dass er *synchron* ist. Das bedeutet, dass man nach dem Aufruf einer *raiseInThread*-Operation sicher ist, dass der betroffene Thread die Exception bearbeitet hat. Für eine Übersicht über die Exceptions siehe Listing 1.4 in Kapitel 5.5.

d) myThreadId

```
myThreadId :: IO ThreadId
```

Mit `myThreadId` kann man die ID des aktuellen Threads, also des Threads, der diese Funktion auswertet, bestimmen. Die Funktion liefert die ID des aktuellen Threads zurück.

Achtung: Wenn man eine Referenz zu einer Thread-ID hat, hat man gleichzeitig eine Referenz zu dem Thread selbst, was verhindert, dass dessen Speicher garbage collected werden kann. Dies soll in einer späteren ghc-Version behoben werden.

e) threadDelay

```
threadDelay :: Int -> IO ()
```

Um die Ausführung eines Threads für einige Zeit zu suspendieren, ist die Funktion threadDelay gedacht. Als Parameter akzeptiert sie die Zeit, die der Thread suspendiert werden soll in Mikrosekunden.

f) yield

```
yield :: IO ()
```

Der Aufruf von `yield` initiiert einen context-switch[12], d. h.: die Ausführung wird in einem anderen Thread fortgesetzt. Dieser Befehl ist in kooperativen Multitasking-Systemen wie hugs wichtig, da sonst ggf. einige Threads ihre Ausführung nicht fortsetzen können. Da der ghc preemptives Multitasking unterstützt, hat dieser Befehl dort keine hohe Bedeutung.

3.2 Kommunikation und Synchronisation zwischen Threads

Concurrent Haskell bietet Variablen mit exklusivem Zugriff, Semaphoren und Channels zur Synchronisation und Kommunikation zwischen Threads.

a) MVars

MVars sind shared variables, in denen Werte abgelegt werden können, auf die andere Threads zugreifen können. Sie sind entweder leer oder mit einem Wert gefüllt. Der Versuch aus einer leeren MVar zu lesen blockiert, bis ein anderer Thread einen Wert in der MVar ablegt.

i) newMVar, newEmptyMVar

Um eine MVar zu erstellen, muss eine der Funktionen `newMVar` oder `newEmptyMVar` aufgerufen werden.

12 Siehe für die Erklärung der Begriffe in Multitasking.-Systemen die Quelle [ASG] oder für eine Kurzdefinition [LHE93].

```
newMVar :: a -> IO (MVar a)
mvar <- newMVar val
newEmptyMVar :: IO (MVar a)
mvar <- newEmptyMVar
```

Beide Funktionen liefern beim Aufruf eine neue MVar in `mvar` zurück. `newMVar` muss noch ein Startwert (`val`) übergeben werden, der in der MVar von Anfang an liegen soll.

ii) takeMVar, readMVar

Um von MVars zu lesen, gibt es zwei Möglichkeiten:. `takeMVar` und `readMVar`.

```
takeMVar :: MVar a -> IO a
readMVar :: MVar a -> IO a
```

Beide Funktionen blockieren, bis sie einen Wert aus der übergebenen MVar lesen können. Dieser wird dann als Ergebnis zurückgeliefert. `takeMVar` entfernt allerdings beim Lesen den Wert aus der MVar. `readMVar` belässt den Wert in der MVar und liefert nur ihren Inhalt zurück. Wenn zwei Threads via `takeMVar` auf dieselbe MVar suspendieren, dann erhält bei einem Schreibzugriff durch einen dritten Thread nur einer der beiden erstgenannten Threads den Wert und der andere Thread blockiert weiter. Wenn ein Thread via `takeMVar` und einer via `readMVar` auf dieselbe MVar suspendieren, gibt es zwei Möglichkeiten bei einem Schreibzugriff durch einen dritten Thread: Entweder beide Threads erhalten den Wert oder nur der mit `takeMVar` suspendierende Thread erhält den Wert, je nachdem welchem vom Scheduler der Lesezugriff als erstem gestattet wird. Welcher Thread somit den Wert zugewiesen bekommt ist unbestimmt. Die Garantien einer MVar beschränken sich darauf, dass bei zwei Zugriffen mit `takeMVar` nur genau ein Thread den Wert erhält.

iii) putMVar

Um einen Wert in eine leere MVar zu legen, muss der Befehl`putMVar` benutzt werden.

```
putMVar :: MVar a -> a -> IO ()
```

Dies schreibt einen Wert vom Typ`a` in die übergebene MVar, falls sie leer ist.

Der Versuch in eine volle MVar zu schreiben löst eine Exception aus. Auch hier ist garantiert, dass nur ein Thread die Möglichkeit bekommt, in eine leere MVar zu schreiben.

iv) swapMVar

Mit `swapMVar` können die Aktionen `takeMVar` und `putMVar` von einem Thread exklusiv (atomar) hintereinander ausgeführt werden.

```
swapMVar :: MVar a -> a -> IO a
```

`swapMVar` suspendiert auf eine MVar bis dort ein Wert anliegt und sie ihn exklusiv lesen darf. Sie nimmt diesen Wert und schreibt direkt anschließend einen Wert zurück, ohne dass ein anderer Thread währenddessen auf die Variable schreiben darf. Nach außen hin erscheint die MVar also über die ganze Operation hin als belegt. Der alte Inhalt wird in zurückgeliefert und der neue wird in die MVar geschrieben.

v) isEmptyMVar

```
isEmptyMVar :: MVar a -> IO Bool
```

Diese Funktion liefert einen Schnappschuss des Zustandes der übergebenen MVar. Ist die MVar leer, so ist liefert die Funktion`True` ansonsten `False`.

Dies besagt nicht, dass im Moment der Auswertung dieses Ergebnisses die MVar nicht bereits wieder belegt bzw. doch schon wieder leer ist. Der Befehl kann aber zum Pollen einer MVar benutzt werden.

b) Semaphoren

Ein häufig eingesetztes Verfahren zur Synchronisation von Threads sind Semaphoren. Mit ihnen kann überprüft werden, ob ein oder mehrere andere Threads eine bestimmte Stelle erreicht haben, um sicherzugehen, dass bestimmte Operationen vor anderen durchgeführt wurden (es gibt auch viele andere Anwendungen, die aber für diese Arbeit nicht relevant sind, siehe dafür [ASG] und [JGF96]).

Semaphoren in Concurrent Haskell speichern Integerwerte. Diese werden beim Aufruf von `waitQSem` erhöht und beim Aufruf von `signalQSem` erniedrigt.

In Concurrent Haskell sind Semaphoren über MVars realisiert.

i) newQSem

```
newQSem :: Int -> IO Qsem
```

Mit `newQSem` kann eine neue Semaphore allokiert und mit dem übergebenen Integer initialisiert werden. Die Funktion liefert die initialisierte Semaphore zurück. Normalerweise wird man die Semaphore mit Null initialisieren.

ii) waitQSem

```
waitQSem :: QSem -> IO ()
```

`waitQSem` erhöht den Wert, der in der übergebenen Semaphore steht um eins und blockiert solange bis in der Semaphore wieder eine Null steht.

iii) signalQSem

```
signalQSem :: QSem  -> IO ()
```

`signalQsem` erniedrigt den Wert in der übergebenen Semaphore und kann somit Threads, die mit `waitQSem` warten, weiterlaufen lassen.

Der exklusive Zugriff beim Schreiben (also beim Erhöhen mit `waitQSem` und Erniedrigen mit `signalQSem`) ist garantiert.

c) Channels

Eine sehr elegante Möglichkeit zur asynchronen Kommunikation zwischen Threads ist die Kommunikation über Channels. Channels sind Puffer bzw. Queues, in denen von Threads Daten abgelegt und von anderen Threads (oder auch demselben) Daten entnommen werden können. Dabei ist sichergestellt, dass ein Datum aus dem Channel, nur an einen Leser abgegeben wird. Wenn eine Leseanfrage auf einen leeren Channel gestellt wird, so blockiert diese bis ein Datum vorhanden ist (analog zum Lesezugriff auf MVars). Auch Channels sind über MVars implementiert.

Um sich das Konzept ein wenig zu verdeutlichen, betrachte man kurz die folgende Bilderserie

Als erstes schreibt Thread 1 das Datum »1« in den Channel. Da der Channel bis jetzt leer ist, wird es an der ersten Stelle im Channel gespeichert.

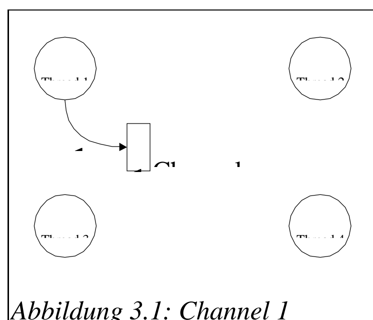


Abbildung 3.1: Channel 1

Anschließend schreibt Thread 2 das Datum »2« in den Channel. Dieses wird vor das Datum »1« gestellt da es neuer ist und somit erst nach dem Datum »1« ausgelesen werden darf.

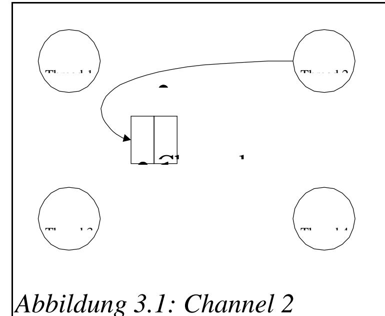


Abbildung 3.1: Channel 2

Nachdem beide Daten in den Channel geschrieben wurden, versucht Thread 3 Daten aus dem Channel zu lesen. Er erhält nach dem FIFO-Prinzip das älteste Datum: »1«. Dieses wird aus dem Channel entfernt. Hätte Thread 3 bereits versucht Daten zu lesen, bevor Thread 1 Daten in den Channel gelegt hätte, so hätte er blockiert, bis das erste Datum (hier die »1« von Thread 1) in den Channel gelegt worden wäre.

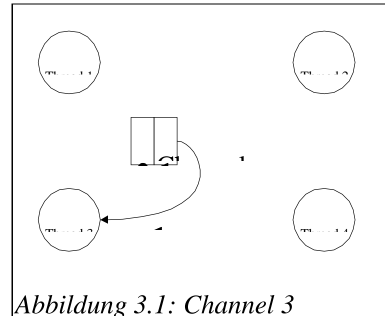


Abbildung 3.1: Channel 3

Als letzter liest Thread 4 Daten aus dem Channel. Er erhält das Datum »2«. Hätten Thread 3 und 4 gleichzeitig versucht, Daten aus dem Channel zu lesen, hätte auch jeder ein anderes Datum bekommen. Allerdings hätte es auch passieren können, dass Thread 4 das Datum »1« bekommen hätte und Thread 3 das Datum »1«. Dies wäre allein davon abhängig gewesen, welchen Thread der Scheduler als erstes den exklusiven Zugriff auf den Channel hätte reservieren lassen.

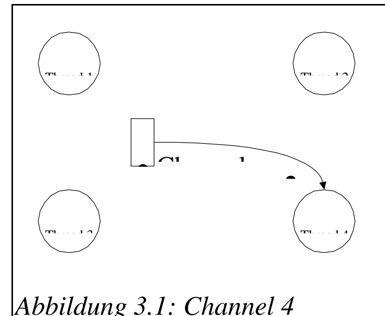


Abbildung 3.1: Channel 4

Concurrent Haskell stellt Funktionen zur Verfügung, um einen neuen Channel zu kreieren, um ihm Daten zu entnehmen und ihm Daten hinzuzufügen.

i) newChan

```
newChan :: IO (Chan a)
```

`newChan` liefert einen neuen leeren Channel zurück.

ii) writeChan

```
writeChan :: Chan a -> a -> IO ()
```

Diese Funktion legt einen übergebenen Wert in einen übergebenen Channel.

iii) readChan

```
readChan :: Chan a -> IO a
```

Die Funktion `readChan` versucht, aus einem übergebenen Channel ein Datum zu entnehmen. Sind keine Daten in dem Channel vorhanden, suspendiert dieser Befehl so lange, bis ein anderer Thread Daten in diesem Channel ablegt. Das Datum wird dann zurückgeliefert.

iv) unGetChan

```
unGetChan :: Chan a -> a -> IO ()
```

Mit `unGetChan` soll ein Wert `val` an das andere Ende im Channel `chan` zurückgelegt werden, so das er beim nächsten Lesezugriff wieder als erstes ausgelesen wird.

Zur Erläuterung betrachte man die nebenstehenden Bilder.

Nachdem Thread 1 und Thread 2 wie in und »1« und »2« in den Channel geschrieben und Thread 3 in Abbildung die einen Wert (die »1«) aus dem Channel gelesen hat, befindet sich das System im nebenstehenden Zustand.

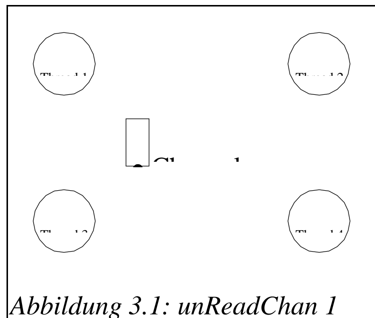


Abbildung 3.1: unReadChan 1

Führt Thread 3 nun eine unread-Operation mit »1« auf den Channel aus (wie rechts angedeutet), so wird die »1« nicht von links in den Channel zurückgeschrieben, sondern von rechts.

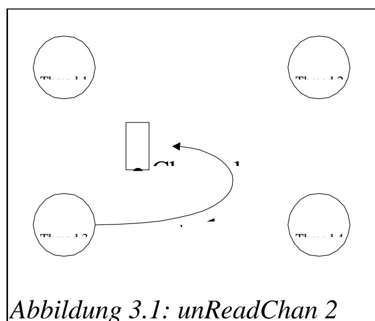


Abbildung 3.1: unReadChan 2

Sie wird also bei der nächsten Leseoperation vor der »2« gelesen. Bei einem normalen writeChan, wäre sie wie in von links in den Channel geschrieben worden.

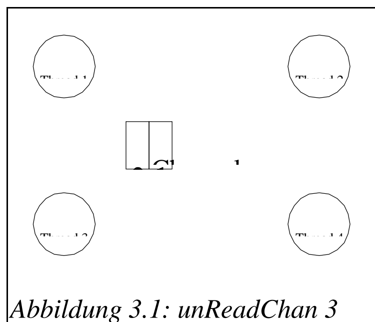


Abbildung 3.1: unReadChan 3

Leider funktioniert diese Routine nicht, wenn mehrere Threads gleichzeitig ein `unGetChan` durchführen.

In der Haskell-Library-Dokumentation ([HLi]) werden einige Befehlsnamen falsch angegeben: Die Schreiboperation heißt dort `putChan` und die Leseoperation `getChan`.

3.3 Zusammenfassung

Concurrent Haskell bietet ein funktionierendes Konzept zur nebenläufigen Programmierung.

Dieses Konzept ist gut geeignet, um es als Ausgangsbasis für eine Erweiterung zur verteilten Programmierung zu benutzen. Ob diese Konzepte selbst zur prozessübergreifenden Kommunikation erweitert werden, oder nur benutzt werden um ein anderes Konzept zu implementieren, bleibt noch zu klären. Das nächste Kapitel wird zeigen, dass ein anderes Konzept sich eher zur Implementierung anbietet.

4. Portbasierte Kommunikation

In Kapitel 2 wurden die Möglichkeiten zur Kommunikation und Programmierung verteilter Systeme in einigen anderen funktionalen Programmiersprachen vorgestellt und in Kapitel 3 wurde ein Konzept für nebenläufige Kommunikation und Programmierung in Haskell vorgestellt, welches sich als Grundlage für eine verteilte Erweiterung von Haskell anbietet.

Dieses Kapitel soll die vorgestellten Konzepte vergleichen und ihre Vorteile und Schwächen in Hinblick auf eine Verwendungsmöglichkeit zur Implementierung einer robusten Erweiterung zur verteilten Kommunikation in Haskell testen. Anschließend soll ein Konzept entwickelt werden, welches sich für eine solche Erweiterung eignet.

4.1 Vergleich der vorgestellten Konzepte

In allen Konzepten gibt es Methoden, um eine asynchrone Kommunikation durchzuführen. Bei all diesen unterschiedlichen Formen werden Daten über oder an eine Struktur geschickt, sei es eine Mailbox in Erlang bzw. Erlang-Style Distributed Haskell, Ein- oder Ausgabeströme wie in Eden, freie Variablen oder Ports (bzw. auch Ein- und Ausgabeströme) wie in Goffin und Curry, oder Channels wie in Concurrent Haskell.

Man findet dort zwei unterschiedliche Arten von Strukturen, über die die Kommunikation läuft. Bei der ersten Art können von einer Struktur mehrere Threads lesen, und bei der zweiten nur ein Thread. Jede diese Struktur erlaubt potentiell den schreibenden Zugriff von mehreren Threads. Deswegen wird die Struktur, die mehrere Leser und Schreiber zulässt, ab jetzt mit »Multiple-Writer-Multiple-Reader-Konzept« (kurz MWMR-Konzept) und die andere, die nur einen Leser und viele Schreiber zulässt, mit »Multiple-Writer-Single-Reader-Konzept« (kurz MWSR-Konzept) bezeichnet. Unter MWSR-Konzept fallen somit das Mailboxkonzept von Erlang und Erlang-Style Distributed Haskell. Alle sind fest an einen Thread gebunden, so dass es bei ihnen höchstens einen Thread gibt, der von ihnen lesen kann.

Alle anderen besprochenen Fälle bieten ein MWMR-Konzept. Auch die constraints in Goffin und Curry bieten die Möglichkeit freie Variablen bzw. Teile aus ihnen (bei unendlichen Listen, sprich Ein- und Ausgabeströmen) in unterschiedlichen Threads auszuwerten und zu binden. Channels sind von ihrer Aufmachung speziell auf ein MWMR-Konzept zugeschnitten (vergleiche Kapitel 3).

Wie sich im Folgenden herausstellen wird, bereiten gerade aber diese MWMR-Konzept-Fälle bei einer robusten Implementierung verteilter Kommunikation große Probleme.

Da mit Concurrent Haskell ein in einem nebenläufigen System gut funktionierendes MWMR-Konzept gegeben ist, wird an dieser Stelle eine Erweiterungsmöglichkeit eines solchen Konzeptes am Beispiel von Concurrent Haskell auf verteilte Systeme betrachtet. Da es um die Untersuchung zur Programmierung robuster verteilter Systeme gehen soll, wird eine Überwachungsinstanz (in den Zeichnungen einfach mit Ü beschriftet) eingeführt, von der angenommen wird, dass sie robust ist, die Systemkomponenten kennt und deren Zustand überprüfen kann. Diese Überwachungsinstanz kann eine oder mehrere Maschinen, aber auch

ein Mensch sein. Sie kann später dann mit Hilfe eines Konzept, dass sich als robust herausgestellt hat, implementiert werden.

Betrachte man einmal die Implementierung eines einfachen Beispiel mit nebenläufiger Kommunikation in Haskell. Es seien mehrere Threads gegeben, die Daten generieren (Writer) und in einen Channel schreiben und mehrere Threads, die Daten aus dem Channel beziehen und verarbeiten (Reader). Es soll jetzt versucht werden, dieses System zu verteilen. Dafür wird nur der worst-case betrachtet, d.h., dass jeder Writer, jeder Reader und auch der Channel selbst auf einem eigenen Rechner laufen. Deshalb wird in den folgenden Abbildungen auf die Darstellung der Prozesse und Rechner verzichtet und nur die Threads (als Kreise, siehe Nomenklatur und verwendete Zeichen, Kapitel 1.3) und der Channel dargestellt.

Bei der Untersuchung dieses Systems, sollen folgende Fragen geklärt werden:

Wie einfach ist das System aufgebaut?

Wie robust ist ein solches System und wie schwer ist es, bei einem Ausfall einer Komponente, das System wieder in einen konsistenten Zustand zu bringen?

Da es sich hier um ein Modell auf sehr betriebssystemnaher Ebene handelt, spiegelt die Betrachtung eine rein imperative Sicht des Sachverhalts wieder. Da die zur Kommunikation über das Netzwerk nötigen Routinen aus der Welt, die durch deterministische und zeitlich bestimmte Abfolge von Ereignissen, stammen, ist diese Betrachtungsweise sinnvoll und angemessen.

a) Verteilung des Multi-Writer-Multi-Reader-Konzeptes von Concurrent Haskell

Die gestrichelte Linie um die Reader und zu den Readern hin soll andeuten, dass die Reader zum Channel eine Verbindung aufbauen, um Daten aus ihm zu nehmen. Im nebenläufigen System bedeutet das, dass die Reader versuchen auf den Speicherbereich des Channels zuzugreifen, um dort Daten auszulesen. Im verteilten System müsste dies durch einen vom Reader aus initiierten Verbindungsaufbau zum Channel und einer Aufforderung an den Channel, Daten zu senden, geschehen. In einem verteilten System muss der Reader dann warten, bis er Daten vom Channel gesendet bekommt. Der Reader muss dem Channel also eine Stelle angeben, wo er im Fall, dass Daten anliegen, diese abgeben kann. Der Reader wartet anschließend, bis die Daten dann dort tatsächlich abgegeben wurden. Dieser Wartezustand soll hier mit »offener Verbindung« bezeichnet werden. Im nebenläufigen System kann der Reader die Daten, sobald vorhanden, einfach entnehmen.

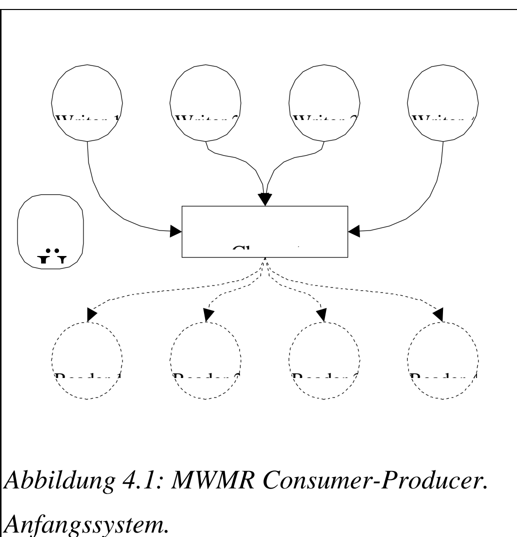


Abbildung 4.1: MWMR Consumer-Producer. Anfangssystem.

Da die Writer die Verbindung zum Schreiben selbst initiieren, sind die Pfeile von den Writern zum Puffer nicht gestrichelt, sondern durchgehend gezeichnet, wie die Umrandung der Writer. Im nebenläufigen System beschreibt ein Writer den Speicher des Channels direkt. Im verteilten System muss der Writer eine Verbindung zum Channel aufbauen und ihm direkt anschließend die zu speichernden Daten übermitteln. Die Verbindung des Writers zum Channel ist also nur zur Zeit der Datenübertragung aktiv. Danach werden keine Daten mehr übertragen, die zu diesem Vorgang gehören. Beim Lesezugriff kann ein solcher Übertragungsvorgang wesentlich länger dauern, da hier im verteilten System ein Reader auf die Antwort des Channels warten

muss. Wenn keine Daten im Channel vorliegen, blockiert der Reader folglich, bis wieder ein Writer Daten im Channel abgelegt hat.

Das System hat den Vorteil, dass die Reader nur bei Bedarf (on demand), also wenn sie freie Kapazitäten besitzen, eine Leseanfrage stellen und somit die von den Writern im Channel abgelegten Aufträge automatisch an einen freien Reader weitergegeben werden.

Wenn in der verteilten Variante dieses Systems ein Rechner ausfällt, auf dem sich ein Writer oder ein Reader aber nicht der Channel befindet, kann das System dadurch wieder in einen konsistenten Zustand gebracht werden, dass der ausgefallene Reader oder Writer wieder neu gestartet werden. Ist ein Reader ausgefallen, der bereits Daten vom Channel angefordert hat, ist dies auch nicht kritisch, da der Channel bei einem Versuch, Daten an diesen Reader zu senden, erkennen kann, dass der Reader nicht mehr existiert und die Daten dann zurück in seine Queue stellen kann, um sie ggf. einem anderen Reader zur Verfügung stellen.

Wenn in einem nebenläufigen System der Rechner ausfällt, auf dem sich der Channel befindet, heißt das, dass gleichzeitig auch alle Prozesse und deren Threads ausfallen. Wenn dies auch eine sehr triviale Beobachtung sein mag, kann man trotzdem sagen, dass sich alle Prozesse, die auf diesem einen Rechner noch laufen, also gar keiner mehr, in einem stabilen Zustand befinden.

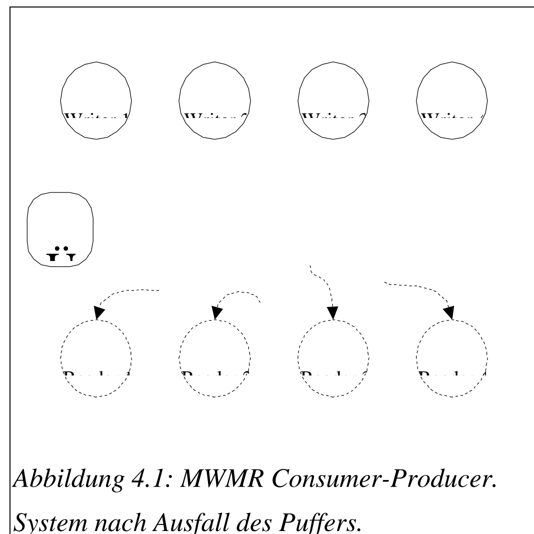


Abbildung 4.1: MWMR Consumer-Producer. System nach Ausfall des Puffers.

Wenn in einem verteilten System der Rechner ausfällt, auf dem der Channel läuft, bleiben alle Reader, die noch auf Daten von diesem Channel warten in einem Deadlockzustand, da sie sich bei dem Channel angemeldet haben und auf eine Antwort warten. Bei einem Ausfall des Channels bleiben also die Verbindungen der Reader, die auf Daten warten, offen.

Die Writer erkennen den Ausfall beim nächsten Versuch auf den Channel zu schreiben, da sie beim Zugriff direkt eine Fehlermeldung bekommen werden. Um weiter Daten senden zu können, müssen sie sich bei der Überwachungsinstanz die Position eines neuen Channels besorgen. Sie können ihre Arbeit also direkt wieder fortsetzen und sind weiter funktionsfähig.

Die Reader können ohne einen komplizierten Mechanismus, der Ihren Deadlockzustand beendet, ihre Arbeit nicht fortsetzen und sind solange nicht funktionsfähig.

Damit das System wieder in einen konsistenten Zustand überführt werden kann, muss die Überwachungsinstanz den Ausfall des Rechners erkennen, auf dem der Channel lief, einen neuen Prozess mit einem Channel Channel' ggf. auf einem anderen Rechner starten, jeden einzelnen Reader über einen besonderen Mechanismus davon in Kenntnis setzen, dass sie ihre offenen Verbindungen frei geben und von dem neuen Channel Daten entgegennehmen. Die Reader müssen einen speziellen Empfangsthread besitzen, der synchron auf eingehende Signale reagieren kann, den alten Leseprozess terminieren und neu starten kann.

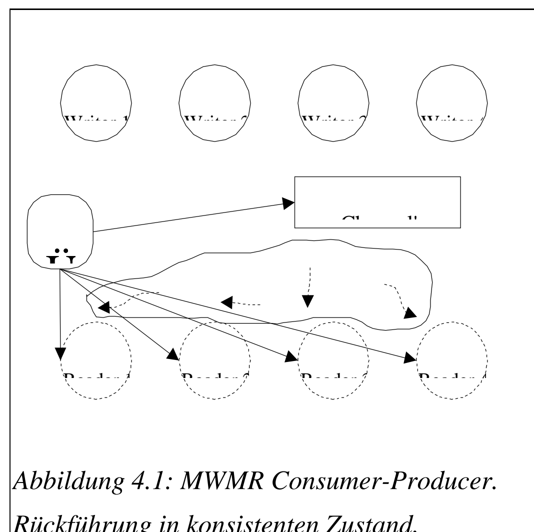


Abbildung 4.1: MWMR Consumer-Producer. Rückführung in konsistenten Zustand.

Erst nachdem die Reader neu gestartet, bzw. sie sich an dem neuen Channel mit einer neuen Leseanfrage angemeldet haben, ist das System wieder konsistent.

Da diese Wiederherstellung mit einem so hohen Aufwand verbunden ist, soll im folgenden Beispiel eine MWMR-Implementierung wie z.B. in Erlang betrachtet und mit dieser Lösung verglichen werden.

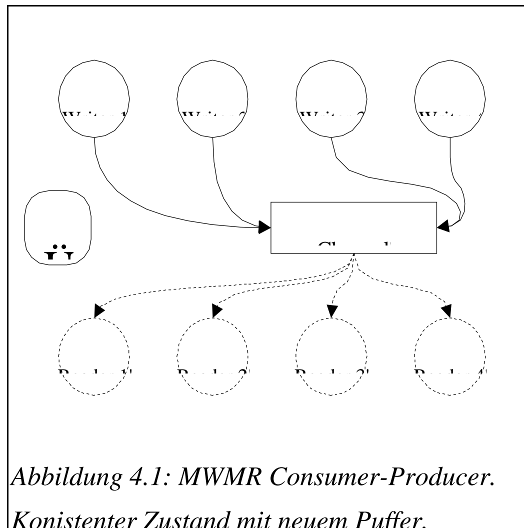


Abbildung 4.1: MWMR Consumer-Producer. Konistenter Zustand mit neuem Puffer.

b) Multi-Writer-Single-Reader Konzeptes für verteilte Systeme

In den folgenden Bildern (bis) soll ein verteiltes System betrachtet werden, in dem zur Kommunikation eine Struktur benutzt wird, die dem MWSR-Konzept genügt. Also eine Struktur, auf die mehrere Threads schreiben, von der aber nur ein Thread lesen darf. Diese Struktur wird im weiteren »Port« genannt. Sie wird in den folgenden Bildern durch einen kleinen schraffierten Kreis mit gestrichelter Umrandung skizziert:. Da nur ein Thread von ihr lesen kann, wird sie direkt an diesen Thread gezeichnet.

Auch hier soll eine Aufgabe wie im vorangegangenen Kapitel verteilt realisiert werden. Mehrere Producer auf unterschiedlichen Rechnern (wieder mit Writer bezeichnet) sollen Aufgaben an einer zentralen Stelle -auch ein eigener Rechner- abgeben können, die von dieser Stelle aus an Worker, die auf unterschiedlichen Rechnern laufen, (wieder mit Reader bezeichnet) verteilt werden sollen. Da in einem MWSR-Konzept jeder Thread, der Daten lesen will, eine eigene Datenstruktur dafür benötigt, da nur er als einziger von dieser Struktur lesen darf, erhält jeder Reader einen Port. Damit die Daten an einer zentralen Stelle abgegeben werden können, ist an Stelle des eben benutzten Channels ein Verteiler nötig, der auch einen Port besitzten muss, an dem die Aufträge abgegeben werden können.

Die Daten, die am Port des Verteilers abgegeben wurden, werden auf die Ports der Reader verteilt. Dieser aktive Zugriff des Verteilerthreads wird wieder durch durchgehende (nicht gestrichelte) Pfeile symbolisiert, da die Aktion der Datenübertragung vom Verteiler ausgelöst wird und nicht von den Readern. Die Reader lesen die Aufträge aus ihren Ports. Liegt der Empfangsspeicher eines Ports in dem Prozess, in dem auch der Reader läuft, zu dem der Port gehört, sind alle Datenübertragungen über das Netzwerk Schreiboperationen, also Operationen, die mit einem sehr kurzen Protokoll auskommen, da nie lange auf Antworten gewartet werden muss.

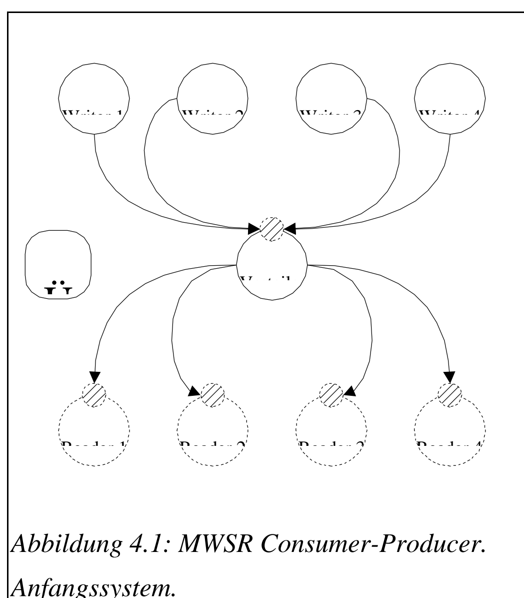


Abbildung 4.1: MWSR Consumer-Producer. Anfangssystem.

Der Ausfall des Channels im letzten Beispiel ist hier mit dem Ausfall des Verteilerprozesses gleichzusetzen. Wenn dieser unvorhergesehen terminiert oder der Rechner, auf dem er läuft, ausfällt, hat dies keine Auswirkung auf die Funktionsfähigkeit der Reader. Die Reader warten zwar auch hier auf Daten, doch können sie die Daten auch von einem anderen Writer bekommen. Die Reader haben keine offenen Verbindungen. Sie können von jedem anderen Writer bedient werden.

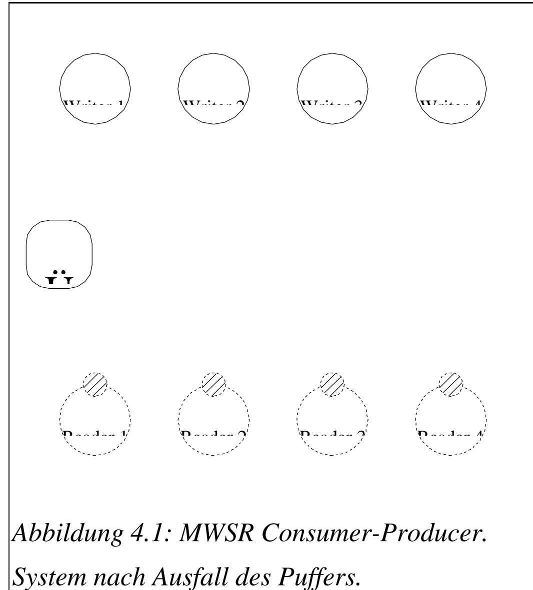


Abbildung 4.1: MWSR Consumer-Producer. System nach Ausfall des Puffers.

Die Writer können sich beim Ausfall des Verteilers genauso wie im letzten Beispiel verhalten. Beim Versuch, dem Puffer Daten zu senden, kann der Thread feststellen, dass dieser nicht mehr existiert und kann dementsprechend reagieren.

Um das System in einen konsistenten Zustand zurückzuversetzen, muss die Überwachungsinstanz die Fehlfunktion des Verteilerprozesses erkennen und diesen ggf. auf einem anderen Rechner (hier unter dem Namen »Channel'«) neu starten. Da der Überwachungseinheit die Reader bekannt sind, kann sie diese dem neuen Verteiler übergeben. Dieser kann dann direkt wieder die Reader bedienen.

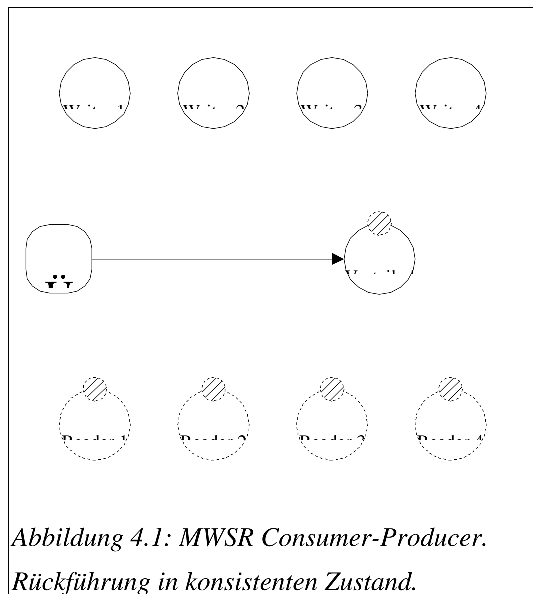


Abbildung 4.1: MWSR Consumer-Producer. Rückführung in konsistenten Zustand.

Die Writer bemerken beim Versuch auf den Port des ausgefallenen Verteiler zu schreiben diesen Ausfall und müssen sich wie im vorangegangen Beispiel bei der Überwachungsinstanz den Port des neuen Verteilers besorgen.

Der Mechanismus zur Wiederherstellung des Systems ist hier also wesentlich einfacher als bei einem MWMR-System.

Ein weiterer Vorteil dieses Systems ist, dass sich die Position des Speicherplatzes der Kommunikationsstruktur, die hier Port genannt wurde, gut festlegen lässt. Sie lässt sich z.B. an den Reader binden. Wenn ein Reader dann ausfällt, fällt auch seine Datenstruktur aus, womit keine offenen Verbindungen entstehen und keine Prozesse, die nicht funktionsfähig sind. Der Ausfall des Readers kann bei einem Schreibzugriff erkannt werden und das System leicht wieder in einen konsistenten Zustand zurückgesetzt werden.

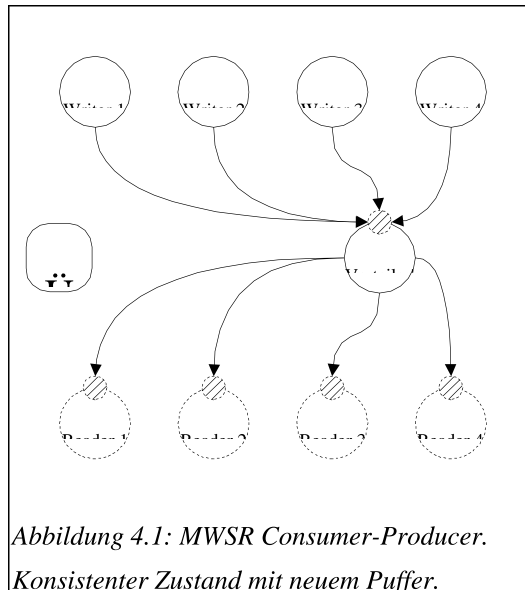


Abbildung 4.1: MWSR Consumer-Producer. Konsistenter Zustand mit neuem Puffer.

Im letzten Beispiel, kann der Channel prinzipiell überall liegen: Auf einem Rechner, auf dem sich eine Reader befindet, auf einem, auf dem sich ein Writer befindet, oder auf einem eigenen Rechner.

Es ist unklar, auf welchem der beteiligten Rechner der Speicherplatz dieser Struktur lokalisiert werden soll. Keine der gerade genannten Varianten ändert etwas an dem vorgestellten Verhalten.

Da in dem in dieser Arbeit entwickelten Konzept die Robustheit und die Reaktionsmöglichkeit, auf instabile Systemkomponenten im Vordergrund stehen sollen, wird der gerade vorgestellte Multi-Writer-Single-Reader-Ansatz als Konzept zur Erweiterung von Haskell eingesetzt.

4.2 Der portbasierte Ansatz

In dem Konzept, um das ich Haskell in dieser Arbeit erweitern möchte, wird eine neue Struktur eingeführt, über die kommuniziert werden kann. Diese Struktur erhält den Namen Port.

Ein solcher Port kann wie die Mailbox in Erlang (Kapitel 2.1) und Erlang-Style Distributed Haskell (Kapitel 2.5) nur von genau einem Thread ausgelesen werden. Ein Thread kann nur von einem Port lesen, den er selber erstellt hat. Ein Port kann von jedem anderen Thread, der diesen Port kennt, also eine Referenz auf ihn besitzt, beschrieben werden. Allerdings kann ein Thread mehrere[13] Ports erstellen, was eher den Ports in Goffin (Kapitel 2.2) und Curry (Kapitel 2.4) ähnelt.

Da es sich um ein Kommunikationskonzept für Haskell handelt, sind die Ports getypt, so dass nur Nachrichten eines bestimmten Typs mit ihnen übertragen werden können.

Um eine asynchrone Kommunikation zu gewährleisten, müssen Daten, die an einen Port geschrieben werden, in diesem Port gepuffert werden. Dafür wird in dem Port eine Datenstruktur gespeichert, die eingehende Daten zwischenspeichern kann. Dadurch ist dieser Puffer immer in dem Thread lokalisiert, der den Port kreiert hat.

Um eine Transparenz zwischen nebenläufigen und verteilten Systemen zu erreichen, müssen sich Ports auch innerhalb eines Prozesses sinnvoll und effizient zur Synchronisation und zum Datenaustausch einsetzen lassen.

Da Ports Schnittstellen zur Außenwelt (zum Netzwerk) sein können, liegen die Funktionen, die auf Ports operieren in der IO-Monade von Haskell.

13 Erlang und Erlang-Style Distributed Haskell hat jeder Thread genau eine Mailbox.

5. Anforderungsdefinition

Dieses Kapitel gibt eine detaillierte Übersicht der nach außen hin sichtbaren Funktionen und Datenstrukturen der in dieser Arbeit entwickelten Erweiterung.

Man braucht eine Funktion, die einen neuen Port erzeugt. Als Name bietet sich hier `newPort` wie in Concurrent Haskell `newChan` an.

Um weiter auf diesen Port zugreifen zu können, ist eine Datenstruktur erforderlich, sinnigerweise mit dem Namen `Port`[14]. Diese sollte von der Funktion, die ihn erzeugt, zurückgeliefert werden.

Die Auswertung von `newPort` liefert dann einen Port zurück, auf den man schreiben, von dem man lesen und den man an andere Funktionen und Threads oder über einen anderen Port auch anderen Prozessen übergeben kann.

Weiterhin sollte es Funktionen geben, mit denen man die eben erwähnten Lese und Schreibzugriffe durchführen kann. Als Namen bieten sich hier in Analogie zu Concurrent Haskell: `readChan` und `writeChan` die Namen `readPort` und `writePort` bzw. **<!>**[15] an. Damit Leseoperationen nicht beliebig lange suspendieren, ist ein Timeout sinnvoll. Zur Vereinfachung des Sendens von Daten an einen extern registrierten Port wäre eine Funktion nützlich, der man den Zielrechner und den Namen eines dort registrierten Ports übergeben könnte. Als Name bietet sich hier `sendToPort` an.

Um nebenläufig mehrere Threads verwalten zu können, sind Funktionen zum Starten und Terminieren von Threads erforderlich. Diese werden aus Concurrent Haskell übernommen. Es handelt sich um `forkIO` und `killThread`.[16] Die einzelnen Prozesse können von Betriebssystemroutinen entweder per Hand oder mit einem entsprechenden Betriebssystemaufruf einer Funktion der ghc-Haskell-Bibliotheken ([HLi]) aufgerufen werden. Für das Starten von Prozessen auf anderen Rechnern soll erst einmal kein Mechanismus implementiert werden. Diese müssen dort per Hand gestartet werden.

Bereits mit den bis hierhin besprochenen Funktionen ist es möglich, nebenläufige Systeme zu programmieren.

Folgendes Beispiel (Listing 5.1) kreiert einen Port, startet einen neuen Thread, der auf diesen Port schreibt und gibt das, was auf dem Port ankommt, auf dem Bildschirm aus:

---

14 Die Wahl des Namen Ports mag ein bisschen unglücklich gewählt scheinen, da der Speicherplatz des Ports ja dort lokalisiert ist, wo er kreiert wurde. Somit ist nur ein Port, von dem man lesen kann, ein »echter« Port. Ports, auf die man nur schreiben kann, sind somit nur Portreferenzen. Um jedoch einen durchgängigen Sprachgebrauch zu erzielen, wird für beide Arten der Begriff Port benutzt, wie bei Concurrent Haskell, wo man auch nur von einem Channel spricht. Der Sachverhalt in Concurrent Haskell entspricht zwar nicht ganz dem des Portkonzeptes, da man auf einen Channel lesen und schreiben kann, doch soll das Portkonzept eine ähnliche Erweiterung wie das Channelkonzept sein, so dass sich ein ähnlicher Sprachgebrauch anbietet.

15 Das <!> ist ein Zugeständnis an das ! in Erlang und an das <!> in Erlang-Style Distributed Haskell.

16 Es werden auch noch weitere Funktionen aus Haskell weitergereicht. Siehe dazu Kapitel 5.5.

```
main =
  do
    p <- newPort
    forkIO (produce p)
    consume p

produce p =
  do
    p <!> 42
    produce p

consume p =
  do
    val <- readPort p
    print val
    consume p
```

Listing 5.1: Einfaches nebenläufiges System

Um anderen Prozessen Zugriff auf diesen Port zu geben, braucht man eine Funktion, die den Port an einer globalen Stelle registriert und eine Funktion, die zu einem Namen auf einem Rechner einen registrierten Port zurückliefern kann. Für diese Funktion werden die Namen `registerPort` und `lookupPort` benutzt.

Fügt man jetzt in Listing 5.1 hinter der Zeile

```
p <- newPort
```

die Zeile

```
registerPort "name"
```

ein, so kann ein anderer Prozess diesen Port mit `lookupPort` erhalten. Unter Annahme, dass der Rechner, auf dem das Beispiel ausgeführt wird, den Namen »Host« hat, kann dann der Port in einem anderen Prozess mit

```
p <- lookupPort "Host" "name"
```

bestimmt werden. Daten können aber auch direkt

```
("Host","name") <!!> data
```

oder

```
SendToPort "Host" "name" data
```

an den Port gesendet werden. Ist »host« der lokale Rechner, so kann man folgendes schreiben:

```
p <- lookupLocalPort "name"
p <!> data
```

Das besprochene Beispiel zeigt, wie ein Prozess mit einem Datenstrom umgehen kann. Was passiert aber, wenn ein Prozess mehrere verschiedene Arten von Daten empfangen soll, z.B. bei einem Chatclient[17]? Dieser muss Daten vom Benutzer über die Tastatur und Nachrichten des Chatservers entgegen nehmen. Fast jeder Prozess eines verteilten Systems mit einer Benutzerschnittstelle braucht diese Fähigkeit. Man kann dieses Verhalten zwar durch mehrere Threads simulieren, dennoch wird es für den Programmierer deutlich einfacher, wenn ihm eine solche Funktion bereits zur Verfügung gestellt wird. Auch die in Kapitel 2 vorgestellten Sprachen bieten diese Möglichkeit[18]. Die Funktion, die zwei Ports zu einem verschmelzen kann, heißt hier: `mergePort` bzw. `<|>`.

In einem Chatclient sieht diese Anwendung dann so aus:

17 Für die komplette Realisierung eines Chatservers und eines Chatclients siehe Kapitel 8.2.

18 In Erlang und Erlang-Style Distributed Haskell wird das Merge bereits automatisch für alle eingehenden Nachrichten ausgeführt, da alle Nachrichten in derselben Mailbox abgelegt werden. Mittels Pattern-Matching wird dann auf einzelne oder eben auf Gruppen von Nachrichten zugegriffen.

```
messageport <- newPort                           -- auf diesem Port sollen Nachrichten empf. werden
chatserverport <!>(Connect messageport)  -- beim Server anmelden
keyboardport <- newPort                          -- auf diesem Port Tastatureingaben empfangen
forkIO (readKeyboard keyboardport)       -- Tastaturleseprozess starten
port <- keyboardport <|> messageport -- Verschmelze die beiden Ports
loop port chatserverport user            -- Werte Eingaben darauf aus
```

Listing 5.2: Zitat aus Chatclient aus Kapitel 8.2.

Insgesamt muss für die Ports berücksichtigt werden, dass nicht immer von einem Port gelesen[19] und auch nicht immer auf ihn geschrieben werden darf. Wie bereits angedeutet, darf man aus implementationstechnischen Gründen bisher nicht auf verschmolzene Ports schreiben. Diese sind nur Konstrukte, um auf mehrere Ports zu suspendieren, also lesend auf sie zuzugreifen.

Eine weitere Forderung an das Projekt ist, dass das Typkonzept von Haskell berücksichtigt werden soll. Also müssen die Ports, wie im letzten Kapitel bereits angedeutet, einen Typ haben. Auch bei der Kommunikation über Netzwerk sollte die Typkonsistenz gewahrt bleiben. Um nicht nur Nachrichten versenden zu können, sondern auch Antworten zu bekommen, sollte es möglich sein, über einen Port einen Antwortport schicken zu können, auf dem eine Antwort übertragen werden kann. Dadurch wäre ein einfacher Mechanismus zur bidirektionale Kommunikation gewährleistet.

Um das Projekt auch für die Programmierung nebenläufiger Systeme benutzen zu können, sollte die Implementation das ganzen so effizient wie möglich sein. D.h., dass eine interne bzw. nebenläufige Kommunikation innerhalb eines Prozesses nicht über das Netzwerk laufen sollte, sondern die Daten in einem solchen Fall, prozessintern direkt über verteilte Variablen binär übergeben werden.

Die Handhabung soll ähnlich einfach wie bei Concurrent Haskell sein, jedoch netzwerkübergreifend transparent. Es soll also nach außen hin keinen Unterschied geben, ob es sich um nebenläufige oder verteilte Kommunikation handelt, so dass eine Erweiterung nebenläufiger Systeme zu verteilten Systemen ohne großen Aufwand möglich ist.

Im Folgenden werden die Datenstrukturen, Konstruktoren und Funktionen, die exportiert werden, um mit Ports in verteilten Umgebungen arbeiten zu können, genau besprochen. Dies ist eine Art Blackbox-Betrachtung des ganzen Projektes. Die anschließenden Kapitel beschäftigen sich dann mit dem inneren Aufbau, der tatsächlichen Umsetzung des Projektes.

5.1 Datenstrukturen

Dieses Kapitel beschreibt die Datenstrukturen, die von dem Projekt exportiert werden.

a) Port

Die wesentliche Datenstruktur dieses Projektes ist `Port t`. Um den Typ `t` über das Netzwerk versenden zu können, muss für ihn eine textuelle Darstellung existieren. Er muss also in Haskell den Klassen `Read` und `Show` angehören.[20] Ggf. könnte man die Daten auch binär über das Netz übertragen, also eine eigene rechnerunabhängige Darstellungsform aller Typen programmieren, doch da die meisten Grundtypen in Haskell bereits den Klassen `Show` und `Read` angehören und sich aus diesen Typen zusammengesetzte Typen auch in die Klassen ableiten lassen, wird in dieser Arbeit die textuelle Darstellung gewählt. Die Übertragung der Daten in anderer Form ist sicherlich ein mögliche Erweiterung des Projektes.

19 Nur der Thread, der den Port kreiert hat, darf von ihm lesen.

20 Es muss also eine eineindeutige Abbildung des Typs auf eine Zeichenkette geben. Die Zeichenkette muss aus einem Zeichenstrom heraus wieder zurückübersetzt werden können.

Da es wünschenswert ist, dass über einen Port auch ein Port verschickt werden kann (siehe Einleitung des Kapitels), über Ports aber nur Typen, die den Klassen `Show` und `Read` angehören, verschickt werden können, muss auch die Datenstruktur Port selbst Mitglied der Klassen `Show` und `Read` sein.

Um eine starke Datenkapselung und somit eine bessere Erweiterbarkeit des Projektes zu erreichen, sollten keine Konstruktoren des Ports exportiert werden. Die Datenstruktur Port muss ein abstrakter Datentyp sein, der nur durch die Anwendung von Schnittstellenfunktionen zurückgeliefert wird.

Man erhält diese abstrakte Datenstruktur als Ergebnis der Anwendung einer der Funktionen `newPort`, `mergePort p1 p2` oder `lookupPort porthost portname`, wobei `porthost` ein Rechnername und `portname` der Name, unter dem ein Port auf diesem Rechner registriert ist, `p1` und `p2` aber wieder Ports sein müssen. Die interne Darstellung dieser Struktur ist in der Implementation gekapselt und wird in Kapitel 6 besprochen.

Von Ports, die mit `newPort` konstruiert wurden, kann der Thread, der diese Funktion aufgerufen hat, lesen, und alle Threads, auch Threads anderer Prozesse, können auf ihn schreiben.

Auf einen Port, der mit `lookupPort` kreiert wurde, kann nur geschrieben werden. Von einem Port, der mit `mergePort` kreiert wurde, darf nur gelesen werden. Selbstverständlich darf auf die Ports, aus denen der Mergeport generiert wurde, geschrieben werden. An die Struktur, die einem ermöglicht, auf zwei Ports Daten zu empfangen, dürfen aber nur über die zugrunde liegenden Ports Daten gesendet werden. Dies hat implementationstechnische Gründe, da es ansonsten zu vieler Verwaltungsthreads bedurft hätte, einen solchen Port zu implementieren. Auch die Frage, wie ein solcher Port verschickt wird und dann wieder aufgelöst wird, ist nicht einfach zu lösen. Dieser Punkt soll bei einer Erweiterung dieses Projektes umgesetzt werden können.

Somit dürfen nur Ports verschickt werden, auf die man auch schreiben darf.

b) PortHost und PortName

`PortHost` und `PortName` sind Datentypen, die zur Spezifikation eines registrierten Ports benutzt werden. Beide werden standardmäßig als `String` definiert. `PortHost` gibt einen Rechnernamen oder eine IP-Nummer an und `PortName` einen beliebigen Namen. Um eine Veränderung der Struktur zur ermöglichen, wird hier nicht direkt `String` verwendet. So könnte man für den Host später auch wirklich nur die IP-Nummer oder gar einen Host-Entry im Sinne einer Posix-Definition speichern wollen. Auch für den Portnamen ist eine komplexere Struktur denkbar. Damit ließen sich gegebenenfalls Konflikte mit mehreren gleichen Portnamen auf einem Rechner ausräumen.

c) Link

Dieser Datentyp dient zur Beschreibung einer Überwachungsverbindung zwischen einem Port und einem Thread. Für eine genauere Beschreibung der Funktionen, die diesen Datentypen erzeugen, siehe Kapitel 5.4.

5.2 Verwaltungsfunktionen

Dieses Kapitel stellt die Verwaltungsfunktionen für Ports vor, die sich mit dem Anlegen und Anmelden von Ports beschäftigen. Alle Verwaltungsfunktionen außer `registerPort` liefern einen Port zurück.

a) newPort

```
newPort :: (Show t, Read t) => IO (Port t)
```

Um einen neuen Port zu erstellen schreibt man:

```
neuer_port <- newPort
```

Diese Funktion legt einen neuen Port vom Typ t an, von dem der Thread, der ihn angelegt hat, lesen darf. Der Port ist für diesen Thread also ein Readport. Dies impliziert natürlich, dass alle Threads auf diesen Port schreiben können.

Ports von denen man lesen und auf die man auch schreiben kann, heißen in Zukunft Readports. Ports, auf die man nur schreiben kann, heißen Writeports und Ports, von denen man nur lesen kann Readonlyports.

Der Typ t des Ports gibt den Typ der Werte an, die über diesen Port verschickt werden können.

Über `registerPort` kann dieser Port anderen Prozessen netzwerkweit zugängig gemacht werden.

b) mergePort, <|>

```
mergePort :: (Show t1, Read t1, Show t2, Read t2) =>
                Port t1 -> Port t2 -> IO (Port (Either t1 t2))
(<|>) :: (Show t1, Read t1, Show t2, Read t2) =>
            Port t1 -> Port t2 -> IO (Port (Either t1 t2))
```

Um einen verschmolzenen Port zu erstellen schreibt man:

```
neuer_mergeport <- mergePort port1 port2
```

oder

```
neuer_mergeport <- port1 <|> port2
```

`mergePort` legt zu zwei Readports (`port1`, `port2`) einen neuen Port an, der einem die Möglichkeit gibt, auf diese zwei Readports gleichzeitig zu suspendieren und die erste eingehende Nachricht auf einem dieser beiden Ports zurückgeliefert zu bekommen.

Über den Either-Typ lassen sich hier Ports beliebiger Typen miteinander verschmelzen. Weiterhin erhält man durch ihn die Möglichkeit, zu bestimmen, auf welchem Port der eingehende Wert ankam. Dabei bedeutet `Left`, dass der Wert auf `port1` ankam, und `Right`, dass er auf `port2` ankam.

Auf den entstehenden Port kann nicht geschrieben werden, nur ggf. auf `port1` und `port2`. Diese Forderung ist technischer Natur und wird Anfang Kapitel 6 erläutert.

c) registerPort

```
registerPort :: Port t -> PortName -> IO ()
```

Um einer Port `port` unter dem Namen `portname` zu registrieren schreibt man:

```
registerPort port portname
```

Mit dieser Funktion wird ein Port anderen Prozessen auf dem Rechner und im Netzwerk zugängig gemacht. Er kann jetzt über den Namen `portname` und den Rechnernamen, auf dem diese Funktion ausgeführt wurde, identifiziert werden. Man beachte, dass diese Funktion keinen neuen Port zurückliefert, sondern der Port durch die Anwendung der Funktion nur die Eigenschaft erhält, direkt von außen identifiziert zu werden. Prinzipiell ist es immer möglich von außen auf einen Port zuzugreifen, auch wenn er nicht unter einem Namen registriert wurde. Dies ist z. B. dann der Fall, wenn der Port über einen anderen Port verschickt wurde.

Ein Name für einen Port kann auf einem Rechner nur einmal vergeben werden. Existiert der Name bereits, so tritt beim Funktionsaufruf eine Exception auf.

d) lookupPort

```
lookupPort :: PortHost -> PortName -> IO (Port t)
```

Diese Funktion ist das Gegenstück zu `registerPort`. Sie liefert zu einem Rechnernamen und einem Portnamen einen Port zurück.

Ist der Port auf dem angesprochenen Rechner nicht registriert, so löst die Funktion eine Exception aus. Ansonsten liefert sie den Port `als Ergebnis`.

e) lookupLocalPort

```
lookupLocalPort :: PortName -> IO (Port t)
```

Diese Funktion ist identisch zu lookupPort, nur dass hier kein Rechnername übergeben werden muss und der lokale Rechner kontaktiert wird.

5.3 Zugriffsfunktionen

Unter den Zugriffsfunktionen sind die Lese- und Schreiboperationen auf die Ports zusammengefasst.

a) readPort

```
readPort :: (Read t, Show t) => Port t -> IO t
```

Um von einem Port `port` einen Wert `value` zu lesen, schreibt man folgendes:

```
value <- readPort port
```

`readPort` erwartet auf einem Port `port` vom Typ `t` einen eingehenden Wert `value` vom Typ `t`. Die Funktion suspendiert so lange, bis ein Wert gelesen werden kann, liest ihn und liefert ihn anschließend in `value` zurück.

b) readPortTimed

```
readPortTimed :: (Read t, Show t) => Port t -> Timeout -> IO t
```

Um von einem Port `port` einen Wert `value` zu lesen, schreibt man folgendes:

```
value <- readPortTimed port timeout
```

`readPortTimed` hat die selbe Funktionalität wie `readPort`, allerdings suspendiert die Funktion nicht unbedingt ewig, wenn kein Wert gelesen werden kann. Wird als Timeout `timeout Nothing` übergeben, so ist das Verhalten der Funktion identisch zu `readPort`. Wird als Timeout `Just count` übergeben, so suspendiert `readPort` nur maximal etwa `count` Mikrosekunden. Wenn bis dahin kein Wert gelesen werden konnte, bricht die Funktion mit einer Exception ab. Ansonsten wird wie beim vorangegangenen Fall ein Wert `value` vom Typ `t` zurückgeliefert.

c) writePort, <!>

```
writePort :: Show t => (Port t) -> t -> IO ()
(<!>) :: Show t => (Port t) -> t -> IO ()
```

Um auf einen Port `port` einen Wert `value` zu schreiben, schreibt man

```
writePort port value
```

oder ähnlich wie in Erlang

```
port <!> value
```

Beide Funktionen schreiben einen Wert `value` vom Typ `t` auf einen Port `port` vom Typ `t`. Wenn die Funktionen auf einen port über Netzwerk (also einen externen Port) schreiben, wartet

diese Funktion auf eine kurze Bestätigungsmeldung, dass der Wert entgegen genommen wurde. Wenn die Gegenseite nicht mehr existiert, tritt eine Exception auf.

d) writePortFast

```
writePortFast :: Show t => (Port t) -> t -> IO ()
```

Diese Funktionen sind analog zu writePort und <!>. Sie warten aber nicht auf eine Bestätigungsmeldung, dass der Wert entgegen genommen wurde. Sie arbeiten somit echt asynchron.[21]

e) sendToPort, <!!>, sendToPortFast

```
sendToPort :: Show t => PortHost -> PortName -> t -> IO ()
(<!!>) :: Show t => (PortHost,PortName) -> t -> IO ()
sendToPortFast :: Show t => PortHost -> PortName -> t -> IO ()
```

Mit diesen Funktionen können Daten direkt an einen registrierten Port gesendet werden. In PortHost wird der Name des Zielrechners und in PortName der Name des gewünschten Ports angegeben. Wenn die Gegenseite nicht mehr existiert, tritt eine Exception auf. <!!> ist identisch zu sendToPort und sendToPortFast arbeitet genau wie writePortFast wieder echt asynchron.

f) sendToLocalPort, sendToLocalPortFast

```
sendToLocalPort :: Show t => PortName -> t -> IO ()
sendToLocalPortFast :: Show t => PortName -> t -> IO ()
```

Diese Funktionen sind identisch zu sendToPort und sendToPortFast, nur dass hier kein Zielrechner übergeben werden muss. Es wird der lokale Rechner kontaktiert.

5.4 Überwachungsfunktionen

Ähnlich wie in Erlang (2.1) gibt es auch in diesem Konzept Funktionen zum Linken und Unlinken mit den Zielen für Schreibzugriffe. Dadurch können in Abhängigkeit von der Existenz von Ports Threads automatisch terminiert werden.

a) link

```
link :: Port t -> IO (Link)
l <- link p
```

link setzt einen Link vom aktuellen Thread zu dem angegebenen Port. D.h.: Der Aufruf dieser Funktion initiiert eine Überwachung, ob der Port noch existiert. Existiert er nicht mehr, wird der Thread, der diese Funktion aufgerufen hat, terminiert. Es wird -im Gegensatz zu Erlang- nicht überprüft, ob mehrere Links von ein und demselben Thread zu demselben Port gesetzt werden.

b) unlink

```
unlink :: Link -> IO ()
unlink l
```

unlink hebt eine mit link gesetzten Link l zur Überwachung wieder auf.

---

21 Dies ist ein spezieller Befehl, dessen Bedeutung an dieser Stelle noch nicht klar zu erkennen ist, da die interne Struktur des Projektes noch nicht bekannt ist. Eine solche Schreiboperation schlägt nur dann fehl, wenn das externe Postamt (siehe Kapitel 6) des angesprochenen Rechners nicht mehr existiert. Ansonsten ist sie immer erfolgreich. Ein Fehler kann dann nur durch einen Timeout auf einem Readport festgestellt werden.

5.5 Weitergereichte Schnittstelle

Die folgenden Funktionen müssen nicht selbst implementiert werden, da sie bereits in den GHC Haskell Libraries ([HLi]) vorhanden sind. Dennoch sollen sie mit in die Schnittstelle aufgenommen werden, um nur noch ein Modul importieren zu müssen, wenn man verteilte Programmierung in seinem Projekt einsetzen möchte.

a) forkIO, killThread, raiseInThread, threadDelay, yield, myThreadId, Threadid

Aus Concurrent Haskell werden die Funktionen `forkIO`, `killThread`, `raiseInThread`, `threadDelay`, `yield`, `myThreadId` und der Datentyp `ThreadId` weitergereicht. Die Beschreibung dieser Funktionen und Datentypen findet man im Kapitel 3 und [HLi].

Aus dem Modul Exception der GHC-Haskell Libraries ([HLi]) wird die Funktion `tryAllIO` als `try` weitergereicht:

b) try

```
try :: IO a -> IO (Either Exception a)
do
  excval <- try f
  case excval of
  ...
```

Mit dieser Funktion kann man die Ausführung einer Funktion `f`, die in der IO-Monade läuft überwachen. Tritt eine Exception in ihr auf, so wird diese als Ergebnis mit vorangestelltem Left zurückgegeben. Trat keine Exception auf, so wird vor das reguläre Ergebnis Right vorangestellt. Das Ergebnis von try wird man meist mit einem `case`-Konstrukt auswerten.

6. Implementation

Dieses Kapitel[22] geht im Detail darauf ein, wie das Projekt realisiert wurde. Dazu gehören die genaue Beschreibung der benutzten Datenstrukturen und der detaillierte Aufbau der einzelnen Module und der internen Schnittstellen.

Nachdem das Verhalten des Systems nach außen hin jetzt festgelegt ist, stellt sich die Frage, was es für Möglichkeiten gibt, dies entsprechend der Anforderungsdefinition (Kapitel 5) algorithmisch umzusetzen.

6.1 Grundlegende Struktur

In diesem Kapitel soll überlegt werden, was für eine Aufteilung bzw. Modularisierung sich für dieses Projekt anbietet.

Auf jeden Fall wird ein Modul benötigt, welches die Schnittstelle nach außen zur Verfügung stellt und die anderen Module bedient. Als Name für dieses Modul bietet sich »Port« an. Das von außen sichtbare Verhalten seiner exportierten Funktionen wurden bereits in Kapitel 5 vorgestellt.

Um Ports unterschiedlicher Prozesse auf einem Rechner zu erreichen, muss es einen ausgezeichneten Prozess geben, der weiß, wie diese Ports zu erreichen sind, da von außen ja immer nur ein fester Punkt angesteuert werden kann. Da dieser ausgezeichnete Prozess Adressen von Ports verwalten muss und somit die externe Kommunikation vereinfacht, wird er das externe Postamt genannt.

Auch intern müssen Ports verwaltet werden. Dies sollte dann analog vom internen Postamt durchgeführt werden.

Somit lassen sich die zur Realisierung des Projektes erforderlichen Funktionen grob in drei Gebiete klassifizieren: Die Verwaltung externer Zugriffe, die Verwaltung interner Zugriffe und das Bindeglied zwischen diesen Verwaltungen und der Schnittstelle.

i) Verwaltung externer Zugriffe

In diesen Bereich gehören:

- Funktionen, die Daten von externen Schreibzugriffen auf Readports entgegen nehmen, die Daten, die intern auf Writeports geschrieben wurden, auf externe Readports schreiben,
- Operationen, die einen Port unter einem Namen registrieren und somit global zugängig machen,
- Operationen, die diesen Namen wieder einen Port zuweisen müssen,
- Funktionen, die die Verbindungen zu anderen Prozessen überwachen.

---

22 Die Überschrift dieses Kapitels lautet bewusst Implementation und nicht Implementierung. Implementierung ist der Vorgang der Entwicklung einer Software. Implementation ist das fertiggestellte Projekt. Da beide in diesem Kapitel beschrieben werden, ist keiner der beiden Titel ganz richtig. Da die Arbeit aber abgeschlossen ist, lautet der Titel Implementation.

Dieser Teil wird »externes Postamt« genannt.

ii) Verwaltung interner Zugriffe

Analog zur Verwaltung externer Zugriffe gehören in den internen Bereich die Funktionen, die intern Daten von Writeports zu Readports übertragen, oder Funktionen, die die Zuordnung von Portdescriptoren zu den Ports übernehmen.

Dieser Teil wird »internes Postamt« genannt.

iii) Bindeglied zur Verwaltung

Um das Ganze nach außen hin, also für den späteren Benutzer, sauber zu kapseln, müssen Funktionen geschaffen werden, die eine Verbindung zwischen den eben beschriebenen Verwaltungen und der in Kapitel 5 festgelegten Schnittstelle schaffen.

6.2 Die Datenstruktur Port

Wie müssen ein Port und der Zugriff auf ihn intern realisiert werden, damit diese nach außen hin die nötige Transparenz haben?

Zur Klärung dieser Frage wird hier erst einmal der genaue Aufbau der Datenstruktur `Port` betrachtet.

Eine wichtige Voraussetzung für Ports war, dass sie einen Typ haben, um das Typkonzept von Haskell und die dadurch resultierenden Vorteile zu wahren. Die Definition von Port muss also folgendermaßen beginnen:

```
data Port t =
```

Dabei gibt `t` den beliebigen Typ an.

Da `Port` eine Struktur ist, um Daten auch ggf. über das Netzwerk zu verschicken,bedeutet dies, dass es möglich sein soll Daten beliebiger Datentypen zu verschicken. Dabei gibt es allerdings ein Problem: Um den Port über ein Netzwerk zwischen unterschiedlichen Rechnern zu versenden, die ggf. eine andere Hardware besitzen und ggf. sogar eine andere Darstellung der internen Datentypen besitzen (z. B. kann ein Integer auf einem Rechner durch 4 Bytes und auf dem andern durch 8 Bytes dargestellt werden), müssen die Daten in eine hardwareunabhängigen Darstellung konvertierbar und wieder aus ihr zurücktransformierbar sein. Da ein eigenes Marshalling-Konzept und ein eigener Precompiler den Umfang dieser Arbeit sprengen würden, muss eine Kommunikation von Rechner zu Rechner über Zeichenketten erfolgen. Dies hat zur Folge, dass die Daten vom Typen der Ports sich in Zeichenketten verwandeln lassen und Zeichenketten sich wieder in Werte dieser Typen zurücklesen lassen können müssen. Die Typen müssen folglich Instanzen der Klassen `Read` und `Show` sein. Deshalb müssen Funktionen definiert werden, die einen Port in eine textuelle Darstellung transformieren und welche, die eine solche Transformation rückgängig machen können.

Um Seiteneffekte zu verhindern, sollte `Port` eine abstrakte Datenstruktur sein. Die Konstruktoren und der Aufbau sollte also von außerhalb nicht sichtbar sein. In der Schnittstellendefinition dürfen nur Port und kein Konstruktor exportiert werden (siehe Beginn des Quelltext Bibliothekslisting 5).

Da Ports nicht aus Konstruktoren erstellt werden dürfen, müssen sie Ergebnisse von Funktionen sein: Wie in der Anforderungsdefinition bereits vorgestellt, sind Ports Ergebnisse der Funktionen `newPort`, `lookupPort` und `mergePort`. (und ihre Vereinfachungen, siehe Schnittstelle in Kapitel 5)

Dennoch liefert jede dieser Funktionen einen Port eines anderen Typs zurück:

Mit `newPort` wird ein Port erstellt, auf den man schreiben und von dem man lesen kann.

`lookupPort` wird dazu benutzt einen extern registrierten Port nachzusehen, auf den man schreiben kann.

`mergePort` liefert die Verschmelzung zweier Ports zurück, von der man lesen können muss. Es ist fraglich, ob es sinnvoll ist, direkt auf diesen Port zu schreiben, da man ja auf seine verschmolzenen Ports schreiben kann. Um auf einen solchen Port direkt schreiben zu können, müsste es für ihn auch eine textuelle Darstellung geben. Das heißt bei der Umwandlung eines solchen Ports müssten irgendwo die Ports dargestellt werden, aus denen dieser Port zusammengesetzt ist. Dafür müssten aber diese beiden Ports, die auch wieder Mergports sein könnten, in der Datenstruktur abgespeichert werden können. Dafür bräuchte man aber die Möglichkeit, in Haskell Listen mit Elementen beliebiger unterschiedlicher Typen darstellen zu können. Da ja durch die beliebige Zusammensetzung von Mergports aus anderen Mergeports ein solcher Port von beliebig vielen unterschiedlichen Typen abhängig wäre. Dies ist aber ohne ein Laufzeittypsystem nicht realisierbar, weshalb ein Verschicken von Mergports nicht möglich ist und deshalb auch interne Schreibzugriffe, die theoretisch realisierbar wären, verboten werden.

So gibt es also drei Typen von Ports:

1. Einen, auf den man schreiben kann und von dem man lesen kann.
2. Einen, auf den man nur schreiben kann.
3. Einen, von dem man nur lesen kann.

Die ersten kann man in den zweiten transformieren, indem man ihn einem anderen Thread übergibt und auch zurücktransformieren, indem man ihn an den Thread übergibt, der ihn erstellt hat. Deshalb liegt es nahe, für die beiden ersten Varianten denselben Konstruktor zu benutzen: `InternalPort`

Die dritte lässt sich in keinen der beiden andern überführen deshalb erhält sie ihren eigenen Konstruktor: `MergePort`

a) Internal Ports

Das Ergebnis von `newPort` ist, wie aus Kapitel 5 bekannt, ein sogenannter Readport.

Da dieser Readport prinzipiell von außen beschreibbar sein muss, braucht er eine Instanz bzw. einen Thread, der diese Daten als Zeichenkette entgegen nimmt und sie in den Typ des Ports umwandelt.

Um asynchrone Schreibzugriffe auf einen solchen Port zu ermöglichen, müssen die ankommenden Daten gepuffert werden. Für diesen Puffer wird einen Datentyp Channel aus Concurrent Haskell (Kapitel 3) vom gleichen Typ wie der Port benutzt, der mit in der Datenstruktur dieser Variante eines Ports gespeichert werden muss.

Bei genauerer Betrachtung fällt auf, dass als Zeichenketten dargestellte Typen von zwei Orten kommen können. Sie können extern oder intern übergeben werden, also können Zeichenkettendaten von außerhalb des Prozesses aber auch von innerhalb des Prozesses übergeben werden. Offensichtlich ist, dass Daten, die von einem anderen Prozess kommen, also über das Netz verschickt werden, als Zeichenketten verschickt werden müssen. Nicht so offensichtlich ist, dass auch intern zwei verschiedene Möglichkeiten vorkommen, wie Daten übergeben werden. Da es noch kein durchgängiges Konzept zur Laufzeittypisierung in Haskell gibt[23], kann es auch nötig sein, dass intern Zeichenketten verschickt werden und keine direkte (binäre) Typübergabe stattfindet.

23 Es gibt bereits Ansätze, die auch für alle Grundtypen funktionieren. Siehe die Bibliothek Dynamic und die Klasse Typable in [HLi]. Leider sind aber noch viele Typen nicht in der Klasse Typable, insbesondere alle Typen von Concurrent Haskell, die für die interne Verwaltung von Ports sehr wichtig sind.

Man betrachte folgendes Beispiel:

```
main =
  do
    p <- newPort
    registerPort(p :: Port Integer) "Test"
    p' <- lookupLocalPort"Test"
    p' <!> (42 :: Integer)
    value <- readPort p'
    print value
```

Listing 6.1: Beispiel für interne Zeichenkettenübergabe

Die interne Darstellung des Ports in der fünften Zeile wird mittels `lookupPort` extern nachgesehen. Hätte man, statt p' durch `lookupPort` zu bestimmen, direkt mit `sendToLocalPort "Test" (42 :: Integer)` auf den Port zugegriffen, würden die Daten sogar über Netzwerkports auf dem lokalen Rechner verschickt. `lookupPort` wandelt diesen externen Port zumindest in einen internen Port um, so dass die Daten, die an diesen Port gesendet werden, dann zumindest intern verschickt werden. Wegen des nicht vorhandenen Laufzeittypsystems ist es leider nicht möglich, alle Ports in einer Datenbank zu speichern, da sie ja beliebige unterschiedliche Typen haben können. Es ist nur möglich Daten eines Typs in einer Datenbank[24] abzulegen. Um die Kommunikation aber dennoch gering zu halten, sollte hier nicht ein Netzwerkport (also ein Port mit Verbindung nach außen) abgelegt werden, sondern eine interne Kommunikationsmöglichkeit. Da über Strings kommuniziert werden muss, bietet sich ein Stringchannel an. Um die Daten von dem Stringchannel in den getypten Channel zu übertragen, ist ein weiterer Thread nötig.

Somit sind für einen Readport zwei Channels nötig, ein Channel eines bestimmten Typs (im weiteren Typchannel genannt), ein Stringchannel für die indirekte Kommunikation (s. o.) und zwei Threads, einer, der die Daten, die von außerhalb des Prozesses auf den Port geschrieben werden, in den Typchannel schreibt und ein Thread, der die Daten aus dem Stringchannel in den Typchannel überträgt.

Da eine Möglichkeit existieren muss, diese Threads für einer Garbagecollection zu terminieren, wird das Protokoll der Ströme von Strings um eine Anweisung zur Terminierung erweitert. Werden keine reinen Daten empfangen, sondern diese Anweisung, terminiert der korrespondierende Thread.

24 Diese Datenbank wird später internes Postamt genannt. Siehe für seine Beschreibung Kapitel 6.4.

zeigt eine schematische Darstellung eines solchen Readports.

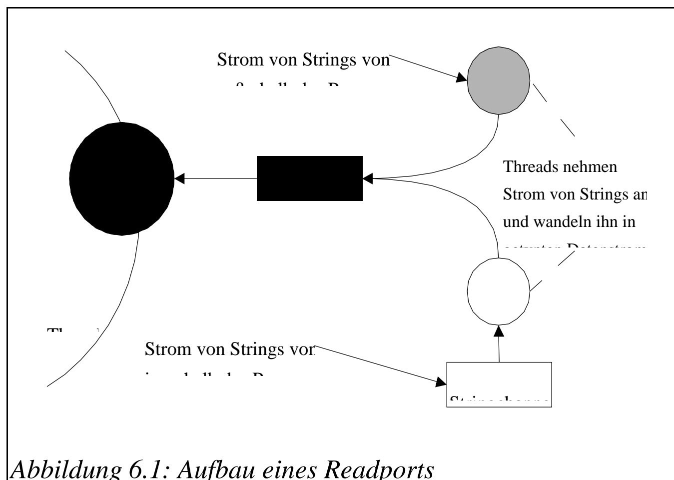


Abbildung 6.1: Aufbau eines Readports

Der Einfachheit halber, wird ein Readport in den weiteren Abbildungen, wie in dargestellt. Dabei stellt der weiße Kreis den Thread, der die internen Strings in den Typchannel schreibt, und den Stringchannel dar und der graue Kreis den Thread, der von außen über einen Netzwerkport Strings entgegen nimmt und sie in den Typchannel schreibt. Das *n* gibt eine Nummer des Readports an, um sie bei mehrfachem Vorkommen unterscheiden zu können.

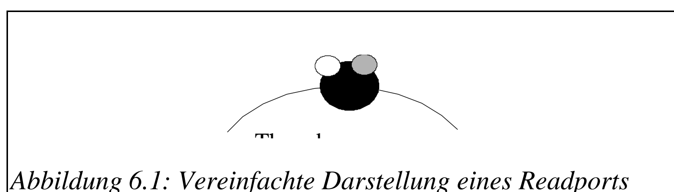


Abbildung 6.1: Vereinfachte Darstellung eines Readports

Eine weitere Forderung an die Struktur Port ist, dass nur der Thread, der ihn angelegt hat, auch von ihm lesen darf. Leider ist es nicht möglich, diese Forderung zur Compilezeit zu überprüfen, wie das folgende kleine Beispiel zeigt:

```
let leser p = do
                value <- readPort p
                print p
                leser p
neuer_port <- newPort
forkIO (leser neuer_port)
```

Listing 6.2: Beispiel für Wichtigkeit der Kontrolle der Thread-ID zur Laufzeit

Die Struktur, die in `neuer_port` steht, wird in keinster Weise dadurch beeinflusst, dass sie an den neuen Prozess übergeben wird. Es lässt sich nicht zur Compilezeit garantieren, dass nicht jemand den Readport mittels `forkIO` an einen anderen Thread übergibt.

Der geneigte Leser mag einwenden, dass es vielleicht schon ausreichen würde, zu fordern, dass nur ein Thread auf den Readport zugreifen darf, und ein Problem bei dieser Variante bisher noch nicht aufgezeigt wurde.

Wenn man aber das Programm, um die Zeile

```
leser neuer_port
```

ergänzt, wird schnell klar, dass sich auch dann nicht garantieren lässt, dass zwei Threads gleichzeitig versuchen, lesend auf den Port zuzugreifen.

So bleibt nichts anderes übrig, als auch die ID des Threads, der den Port mittels `newPort` kreiert in dem Readport zu speichern und diese zur Laufzeit bei einem Lesezugriff zu überprüfen.

Um einen Readport eindeutig im Netzwerk zu identifizieren, muss in ihm eine netzwerkweit eindeutige Port-ID gespeichert werden. Da der Begriff PortID in Haskell aber bereits für die

Kommunikation mit Sockets (Netzwerkports) verwendet wird, die ich auch benutze, werde ich in Zukunft für den Begriff Port-ID »Portdescriptor« verwenden.

Damit dieser Portdescriptor eindeutig ist, besteht er aus der IP-Adresse des Hosts auf dem er erschaffen wurde, der Prozess-ID des kreierenden Prozesses und einer Referenznummer, die innerhalb des Prozesses eindeutig ist (siehe Listing 6.4).

So erhält man für solch einen Port folgende Struktur:

```
data Port t = ...
  InternalPort
  {
    pDesc :: PortDescriptor,
    pInfo :: PortInfo t
  } ...
```

Listing 6.3: Internal Port

mit

```
data PortDescriptor =
  PortDescriptor
    {
      pHost      :: PortHost,  -- IP-Addresse des Hosts als String
      pProcessId:: ProcessID, -- Prozessnummer
      pRefNr     :: Integer    -- Eindeutige Referenznummer
    }
    deriving (Read, Show, Ord, Eq)
```

Listing 6.4: PortDescriptor

und

```
data PortInfo t =
  PortInfo
    {
      -- Channel über den Daten des Ports geschickt und gelesen werden
      pChannel  :: MVar (PortChannel t),
      -- ThreadID des Threads, der von diesem Port lesen darf (der ihn kreiert hat)
      pThreadId :: MVar(Maybe ThreadId),
    }
```

Listing 6.5: PortInfo

Als Konstruktor für den Port wurde `InternalPort` und nicht `ReadPort` gewählt, da dieser Port ja durch Übergabe an einen anderen Thread nicht mehr lesbar ist (also ein Writeport wird).

Um in der Lage zu sein, Ports über Portdescriptoren in Listen zu suchen und zu sortieren, muss die Datenstruktur `PortDescriptor` Mitglied der Klassen `Ord` und `Eq` sein.

`pChannel` und `pThreadId` werden in MVars gespeichert, damit es möglich ist, den Port sauber zu zerstören und seinen Speicher freizugeben.

Damit mit diesem Konzept eine leichte bidirektionale Kommunikation möglich wird, müssen diese Readports über Ports verschickt werden können. Es muss also eine Darstellung des Readports als Zeichenkette geben, die sich an einer anderen Stelle wieder zu einem Writeport zusammensetzen lässt. Für die eindeutige Darstellung eines Ports reicht der Portdescriptor aus, der `PortInfo`-Anteil enthält nur Daten, die für die interne Repräsentation des Ports wichtig sind. Deshalb muss dieser aber eine Instanz der Klassen `Show` und `Read` sein.

`PortChannel` ist aus Effizienzgründen nur dann ein getypter Channel, wenn er über `newPort` erzeugt wurde.

Wenn er über `lookupPort` (also Writeport ist) erzeugt wurde, ist er entweder ein Stringchannel (`PCipoChan`: Channel des internen Postamtes, dies ist ein Stringchannel, der um eine Terminieraufforderung erweitert wurde), falls der zugehörige Readport prozessintern liegt, oder ein Verweis auf einen extern liegenden Socket (`PCSocket`). Wird der Port nicht

mehr verwendet und steht kurz vor der Garbagecollection, steht in `PortChannel PCNothing`.

```
data PortChannel t =   PCTypedChan (Chan t)
                     -- IPOChannelType: Stringchannel um Terminieraufforderung erweitert
                     | PCipoChan (IPOChannelType)
                     -- hier gibt es ein wenig Redundanz zu Gunsten der Übersichtlichkeit
                     | PCSocket PortDescriptor
                     | PCNothing -- Toter Port
```

Listing 6.6: PortChannel

Kurz bevor der Port garbagecollected wird steht in `pChannel PCNothing`, um anzuzeigen, dass der Port eigentlich schon tot ist.

b) Mergeports

Die letzte Forderung an die Struktur Port ist, dass sie in der Lage sein soll, die Verschmelzung zweier Ports zu repräsentieren, also einen Readonlyport, der bei der Anwendung der Funktion `mergePort` entsteht. Um solche Ports darzustellen gibt es den Konstruktor `MergePort`.

Problematisch an dieser Stelle ist, dass in einem Port die zwei in der Regel unterschiedlichen Typen der verschmolzenen Ports gespeichert werden müssen. Dies lässt sich über den Either-Typen von Haskell realisieren. Ein Port der die Verschmelzung zweier Ports mit den Typen `Port t1` und `Port t2` repräsentiert, muss also den Typen `Port (Either t1 t2)` haben. Dies bedeutet aber, dass man die Ports, die verschmolzen wurden, nicht in der Struktur Port mit abspeichern kann. Trotzdem muss dort eine Art Referenz auf die beiden Ports abgespeichert werden, damit die Garbagecollection funktioniert.

Dies wird durch einen kleinen Trick realisiert. Das Problem wird auch wieder von der Compilezeit zur Laufzeit verschoben, indem beim Aufruf von `mergePort` ein Thread angelegt wird, der die beiden verschmolzenen Ports kennt und eine Kommunkationsvariable (`mpMessage`), in der die Struktur des Mergeports gespeichert wird. Weiterhin muss noch eine Variable vom Typ des Ports in der Struktur gespeichert werden (`mpMVar`), in der bei Suspension auf den Mergeport der Wert eines der beiden Ports zurückgegeben wird. Über die Kommunikationsvariable vom Typ `MergePortMessage` lässt sich dann eine Aufforderung zum Terminieren an diesen Thread schicken, oder die Aufforderung, einen Wert eines der beiden Readports entgegen zu nehmen und in der dafür vorgesehenen Variable zu speichern oder die Aufforderung einen Wert zurückzulegen falls zufällig Werte aus beiden verschmolzenen Ports gelesen wurden und somit einer zu viel vorhanden ist. Dieser Datentyp wird in Kapitel 6.4 genauer erläutert.

Es muss garantiert werden, dass nur der Thread, der die beiden Readports verschmolzen und kreiert hat, auch auf den Mergeport suspendieren kann, und nicht gleichzeitig ein anderer Thread auf einen der Readports oder Mergeport suspendieren. Deshalb muss auch beim Mergeport eine Referenz auf den kreierenden Thread gespeichert werden.

So sieht die Struktur für den Mergeport folgendermaßen aus:

```
data Port t = ...
  MergePort
  {
    mpThreadId :: ThreadId,
    mpMVar :: MVar (Maybe t),
    mpMessage :: MVar (MergePortMessage t)
  } ...
```

Listing 6.7: MergePort

c) Die komplette Datenstruktur Port

An dieser Stelle ist es möglich, die komplette interne Datentypdefinition von Port anzugeben.

```
data Port t =
  InternalPort -- Von diesen kann man manchmal lesen (ReadPorts)
  {
    pDesc :: PortDescriptor,
    pInfo :: PortInfo t
  } |
  MergePort -- Diese Ports sind ReadonlyPorts, Verschmelzung zweier Ports
  {
    mpThreadId :: ThreadId,
    mpMVar :: MVar (Maybe t), -- Maybe, da es vielleicht nicht im angeg. Timeout
                              -- geschafft wird
    mpMessage :: MVar (MergePortMessage t)
  }
```

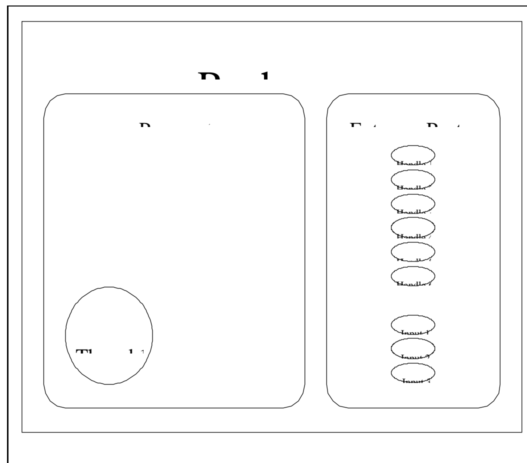
Listing 6.8: Komplette Datenstruktur Port

Die Datenstruktur Port besitzt also zwei Konstruktoren: `InternalPort` und `MergePort`. Diese sind nicht an der Schnittstelle sichtbar. Nur der abstrakte Datentyp wird exportiert.

Damit Ports über das Netz verschickt werden können, muss zu ihnen eine textuelle Darstellung existieren. Da nur auf Ports mit Konstruktor `InternalPort` direkt geschrieben werden kann, existiert eine solche Darstellung auch nur für diesen Typ. Wird versucht, einen Mergeport zu verschicken, tritt eine Exception auf, die nicht vom Typ IO ist. Deshalb kann sie nicht mit den normalen Methoden abgefangen werden.

6.3 Schematische Darstellung der Kommunikation

Dieses Kapitel gibt an einem Beispiel eine schematische Darstellung dessen, was beim Kommunizieren zwischen den einzelnen Modulen passiert.



zeigt das System beim Start des ersten portbasierten Prozesses. Das externe Postamt wurde bereits gestartet. Am Anfang läuft in dem Prozess nur der Hauptthread. In erzeugt Thread 1 via `newPort` einen Readport (Readport 1). Dieser meldet sich beim internen Postamt an, das von da ab eine Referenz auf den Stringchannel hält. Weiterhin meldet er sich beim externen Postamt an und etabliert eine Verbindung zu ihm, d.h.: ab jetzt kann das externe Postamt eine Zuordnung des Portdescriptors des Readport 1 zu der tatsächlichen Verbindung (über den Handle dieser Verbindung) zu dem Port herstellen.

Für jeden Readport hält das externe Postamt einen Handle offen, zu dem es Daten schickt, die an den entsprechenden Port gerichtet sind. Die in Kapitel 6.2 beschriebenen Threads (durch den kleinen weißen und grauen Kreis symbolisert) werden gestartet. Sie nehmen die Daten in Strings entgegen und wandeln sie in getypte Daten um.

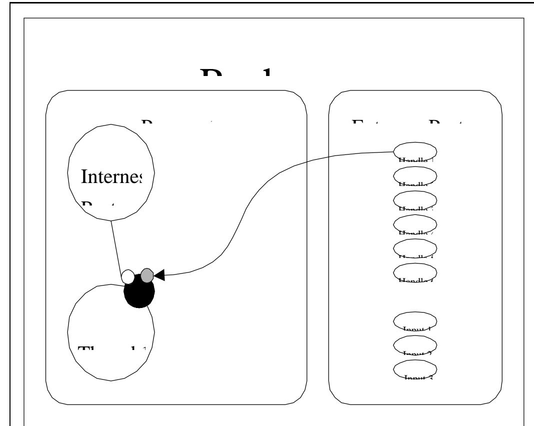


Abbildung 6.1: Thread 1 kreiert Readport 1

An dieser Stelle bleibt zu bemerken, dass ein direktes Senden der Daten an den Port ohne Umweg über das externe Postamt nicht schneller sondern sogar langsamer wäre, da bei jedem Zugriff im externen Postamt die Position (bzw. der Socket) des Readports erfragt werden müsste, um sicherzugehen, dass der Port noch existiert. Es reicht nicht, an die selbe Socketnummer Daten zu schicken, da diese mittlerweile von einem anderen Port belegt sein könnte. Es wäre ggf. möglich, für jede Verbindung Readport-Writeport eine Verbindung offen zu halten, um dieses Protokoll zu minimieren, doch würde damit die Möglichkeit, Ports zu schließen, wenn sie nicht mehr gebraucht werden, verhindert.

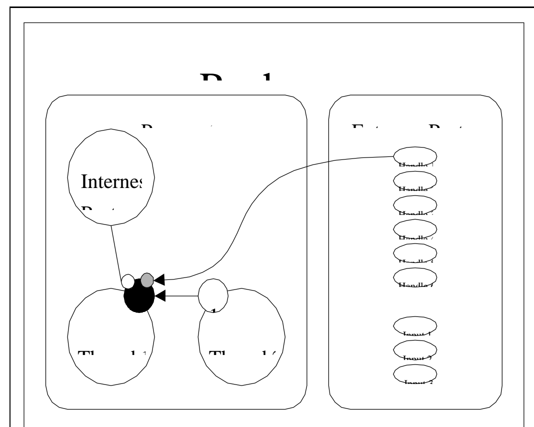


Abbildung 6.1: Start Thread 2, Übergabe Readport, Umwandlung in Writeport

In wird ein zweiter Thread (Thread 2) von Thread 1 aus gestartet. Als Parameter wird ihm der Readport 1 übergeben, der somit für Thread 2 zum Writeport 1 wird. Bei solch einer internen Übergabe eines Ports über Parameter, können Daten direkt von Thread 2 nach Thread 1 übertragen werden, ohne zwischendurch in Strings verwandelt zu werden. Deswegen ist in die Zeichnung auch eine direkte Verbindung eingezeichnet. Daten, die auf den Writeport 1 geschrieben werden, werden direkt in den Typchannel des Readports 1 geschrieben.

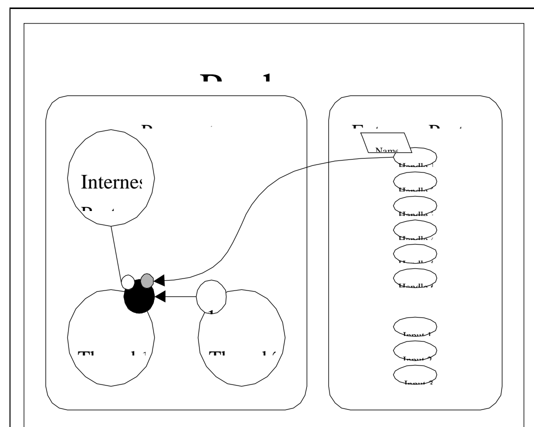


Abbildung 6.1: Registrieren des Readport 1 unter »Name«

In wird Readport 1 unter dem Namen »Name« im Externen Postamt registriert. Dieses kannte bisher nur den Portdescriptor des Ports, so dass Daten extern nur verschickt werden konnten, wenn ein anderer Prozess den Portdescriptor kannte. Jetzt können von außen Daten auch direkt an den Namen geschickt werden, da das externe Postamt die Zuordnung vornehmen kann.

Sie können allerdings nicht nur von außerhalb über diesen Mechanismus verschickt werden. Wenn in Thread 2 aus Prozess 1 via `lookupPort "Name"` im externen Postamt den internen Port 1 nachsieht, kommt es zu den in Kapitel 6.2 beschriebenen Typzuordnungsproblemen. Da das interne Postamt nur eine Datenbank von Stringchannels hält, bekommt dieser Writeport diese Referenz zugewiesen und schreibt somit auf den Stringchannel des Readport 1. Die Daten werden zwar noch intern verschickt, da der Port aber im internen Postamt nachgesehen werden musste, müssen die Daten als Strings verschickt werden. Dies nenne ich die indirekte interne Kommunikation.

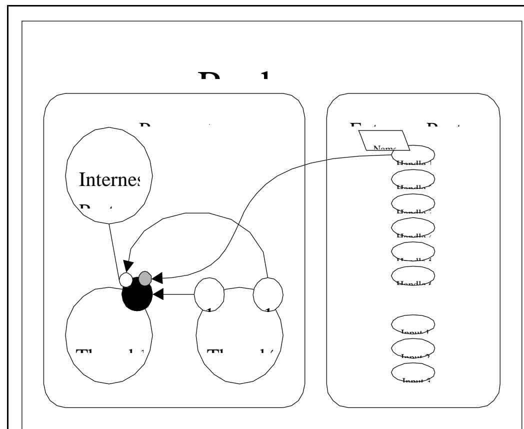


Abbildung 6.1: Registrieren des Readport 1 unter »Name«

zeigt den Start eines zweiten portbasierten Prozesses (Prozess 2) auf Rechner 1 und den in diesem Prozess laufenden Mainthread.

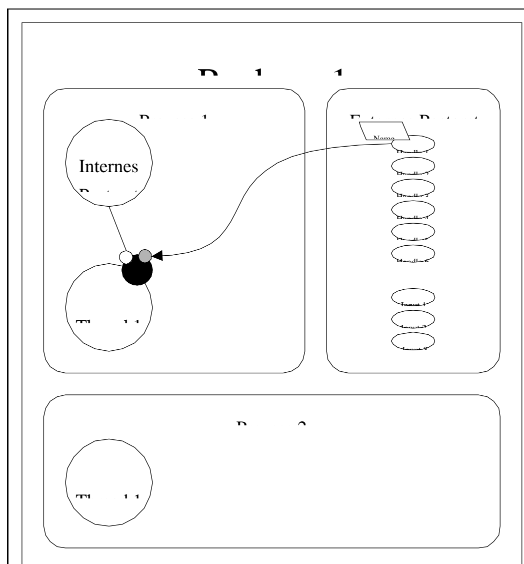


Abbildung 6.1: Prozess 2 auf Rechner 1 wird gestartet

Dieser Prozess legt einen lokalen benannten Writeport an () an und sendet Daten über diesen Writeport ().

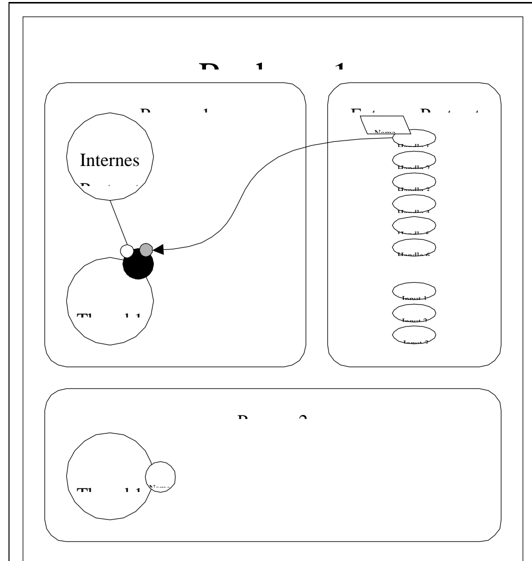


Abbildung 6.1: Thread 1 in Prozess 2 legt benannten Writeport »Name« an

Da es sich um einen *lokalen* benannten Writeport handelt, werden diese Daten und der Name an das lokale externe Postamt geschickt. Da der Name im Postamt registriert ist, kann dieses eine Zuordnung vornehmen und die Daten über die richtige Verbindung weiterleiten. Diese Verbindung ist gestrichelt eingezeichnet, da sie nur so lange aktiv ist, wie Daten auch tatsächlich gesendet werden. Sie wird nach dem Versand direkt wieder geschlossen. Somit ist sichergestellt, dass beim Versuch eines Schreibzugriffs auf einen nicht mehr existenten Readport der Writeport dies auch erfährt.

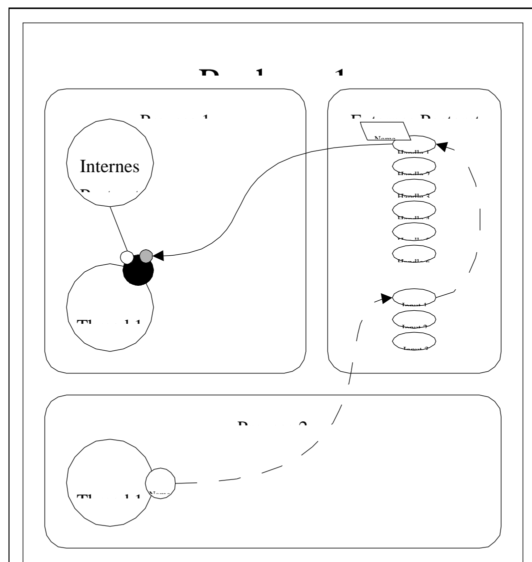


Abbildung 6.1: Thread 1 in Prozess 2 schreibt Daten auf lokalen benannten Writeport »Name«

Ganz analog funktioniert die Kommunikation über Rechnergrenzen hinweg. In unserem Beispiel startet Rechner 2 den Prozess 1 (dies ist nicht derselbe Prozess wie auf Rechner 1, aber der erste portbasierte Prozess auf Rechner 2). Auf ihm wurde bereits ein externes Postamt gestartet ( ).

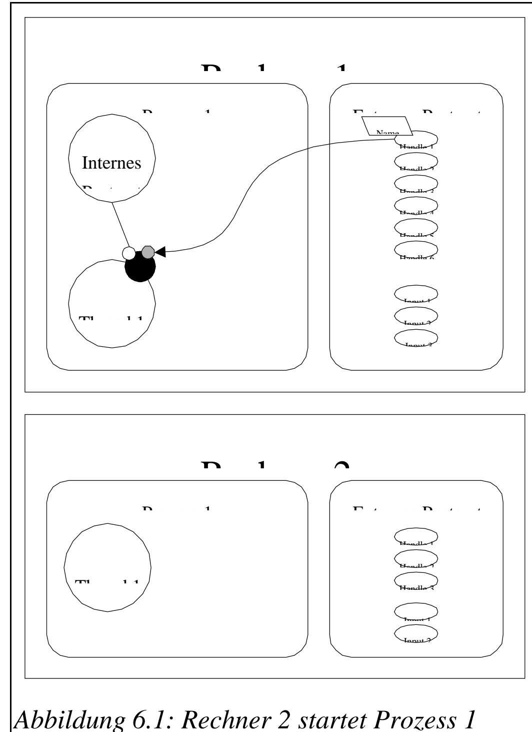


Abbildung 6.1: Rechner 2 startet Prozess 1

Anschließend wird auch hier wie bei der lokalen prozessübergreifenden Kommunikation () ein benannter Writeport generiert, nur ist es hier kein lokaler benannter Writeport, sondern der Name des Zielrechners ist mit in ihm kodiert.

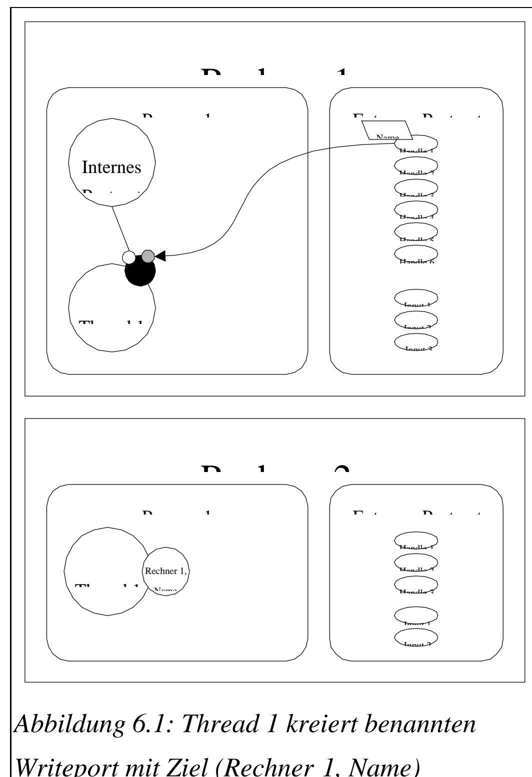


Abbildung 6.1: Thread 1 kreiert benannten Writeport mit Ziel (Rechner 1, Name)

Das Schreiben der Daten ist völlig analog zum Schreiben zu lokalen Prozessen (), außer dass eine Verbindung zum externen Postamt des Zielports aufgenommen wird ().

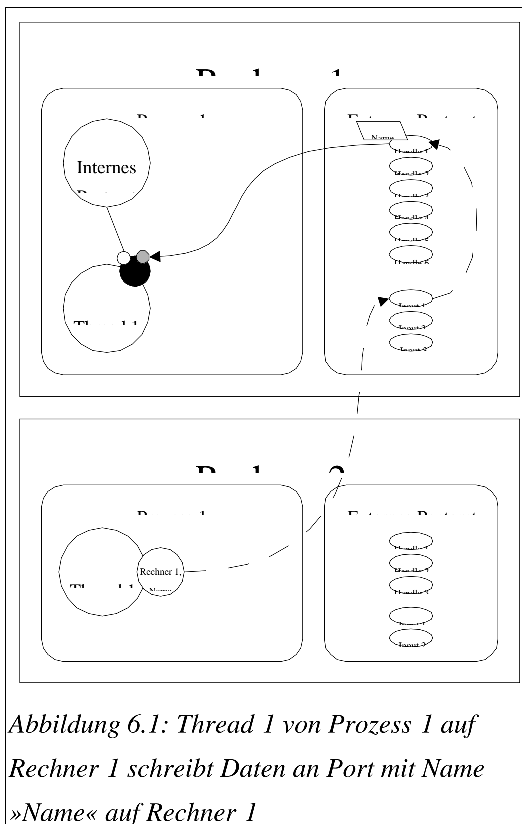


Abbildung 6.1: Thread 1 von Prozess 1 auf Rechner 1 schreibt Daten an Port mit Name »Name« auf Rechner 1

6.4 Die Module der Bibliothek

Wie schon in Kapitel 6.1 erwähnt, besteht die Bibliothek im Wesentlichen aus drei Teilen:

Dem externen Postamt, dem internen Postamt und der Portschnittstelle. Hinzu kommt noch ein kleines übergeordnetes Modul mit gemeinsamen Datenstrukturen und Hilfsfunktionen namens PortGlobals.

Die Implementationen der einzelnen Module werden nun der Reihe nach beschrieben, um die interne Funktionsweise der Bibliothek zu dokumentieren.

Im letzten Abschnitt wird kurz auf das Makefile eingegangen, welches diese Bibliothek übersetzt.

a) Globale Datenstrukturen und Hilfsfunktionen (PortGlobals.hs)

In diesem Modul werden Funktionen und Datenstrukturen definiert, die alle anderen Module benutzen. Deshalb muss auch die Datei, die diese Schnittstelle beschreibt (PortGlobals.hi) zusammen mit der erstellten Bibliothek kopiert werden, um auf die Funktionen aus dem Modul Port.hs zugreifen zu können.

Für den kompletten Quelltext von PortGlobals siehe Anhang B, Bibliothekslisting 1.

Die sicherlich wichtigste Datenstruktur dieses Moduls ist PortDescriptor. Sie wurde bereits in Kapitel 6.2 vorgestellt und in Listing 6.4 abgedruckt. Wie schon erwähnt, besteht dieser Portdescriptor aus der IP-Adresse des Hosts auf dem er erschaffen wurde, der Prozess-ID des kreierenden Prozesses und einer Referenznummer, die innerhalb des Prozesses eindeutig ist. Der Typ des Hosts (`PortHost`) und des Namens (`PortName`) werden auch in diesem Modul festgelegt. Sie sind beide als `String` definiert (vergleiche Quelltext). Die IP-Adresse hätte man zwar auch aus vier Bytes zusammensetzen können, da aber in Host prinzipiell auch ein Rechnername stehen kann, habe ich `String` als Datentyp gewählt. Um ihn dennoch später

durch einen anderen Typen ersetzen zu können, ist er hier als eigener Typ definiert worden. Die Prozess-ID wird aus dem Posix-Modul (siehe [HLi]) importiert.

Zwei Datentypen werden in diesem Modul definiert, die eigentlich in das Modul IntPostOffice gehören. Es war nötig, diese beiden in dieses Modul mit aufzunehmen, um nicht noch ein dritte Schnittstellenbeschreibungsdatei an die Bibliothek zu binden[25]. Es handelt sich um `IPOTransferMessage` und `IPOChannelType`. Da sie eigentlich ins IntPostOffice gehören, werden sie auch erst dort beschrieben.

Neben den eben besprochenen Datenstrukturen stellt das Modul PortGlobals noch ein paar Hilfsfunktionen zur Verfügung, die von allen Bilbliotheksbausteinen verwendet werden.

`myIPAddress` liefert die IP-Adresse des Rechners, auf dem diese Funktion aufgerufen wird im Datentyp `PortHost` zurück. myIPAddress liegt selbstverständlich in der IO-Monade, da ihre Ausführung je nach Rechner, auf dem diese Funktion ausgeführt wird, einen anderen Wert liefert.

`nop` ist die leere Operation. Sie wird z. B. benötigt, um in einer imperativen `if-then-else`-Anweisung im `else`-Zweig nichts zu tun oder um eine Funktion abzuschließen, die sonst mit dem Befehl `forkIO` geendet hätte.[26]

`forget` ist eine Funktion, die die Auswertung einer Funktion der IO-Monade ermöglicht, ohne tatsächlich ihr Ergebnis auszuwerten.

b) Externes Postamt - External Post Office (ExtPostOffice*.hs, NSocket.hs)

Das externe Postamt (External Post Office, abgekürzt EPO oder epo) ist ein eigenständiger Prozess, der auf jedem Rechner, auf dem dieses Projekt eingesetzt werden soll, genau einmal[27] laufen muss. Er muss manuell[28] auf einem Rechner gestartet werden, bevor irgendein Port in einem anderen Prozess angelegt werden kann.

Zum externen Postamt gehören ein Modul mit neuen Socketzugriffsfunktionen, eins mit Konstanten für die externe Kommunikation und dem externen Kommunikationsprotokoll, das Prozessmodul selbst und ein Modul mit Funktionen, um von außen auf das externe Postamt zuzugreifen.

i) Konstanten mit Kommunikationsprotokoll, Timeouts und Socketnummer (ExtPostOfficeConstants.hs)

Im Modul ExtPostOfficeConstants werden einige Konstanten und zwei Kommuniktaionsprotokolle mit dem externen Postamt definiert. Für den kompletten Quelltext von ExtPostOfficeConstants siehe Anhang B, Bibliothekslisting 3.

---

25 Zwei Schnittstellendateien sind bereits unschön genug, aber an dieser Stelle mussten ein sauberes Modulkonzept gegenüber einer vernünftigen Datenkapselung abgewägt werden. Leider ist der ghc (siehe User-Manual des ghc in [GHC]) auf das Interface eines Moduls angewiesen, wenn ein in einem Obermodul exportierter abstrakter Datentyp einen Datentyp des importierten Moduls benutzt, obwohl der importierte Datentyp nicht explizit im Obermodul exportiert wird. Für eine nähere Beschreibung dieses Problems siehe Kapitel 6.5 (Sprachspezifische Probleme).

26 Auch hierbei scheint es sich um eine kleine Ungereimtheit im ghc zu handeln. Näheres dazu im Kapitel 6.5 (Sprachspezifische Probleme).

27 Es ist zwar prinzipiell möglich, mehrere externe Postämter auf einem Rechner zu starten, allerdings muss dafür die Socketnummer (`epoPortNumber` / default 5600), an die sich das Postamt bindet, in ExtPostOfficeConstants.hs geändert und das gesamte Projekt neu compiliert werden.

28 Manuell heißt in diesem Fall nicht unbedingt, dass man dies per Hand tun muss. Eher, dass es außerhalb des Haskellprogramms mit Mitteln des Betriebssystems geschehen muss. Selbstverständlich ist es möglich, diesen Prozess als Dienst oder mittels einer Verknüpfung in einem Autostartordner automatisch ausführen zu lassen.

Die beiden Konstanten sind `epoPortNumber` und `epoLifeCheckTime`.

`epoPortNumber` ist die Nummer des Sockets, auf dem das externe Postamt auf Anfragen von externen Prozessen horcht. Es ist von Typ PortID[29] und standardmäßig als `PortNumber 5600` definiert. Um das externe Postamt auf einem anderen Socket horchen zu lassen und um die Bibliothek sich mit dem externen Postamt auf einem anderen Socket verbinden zu lassen, muss die 5600 in eine andere Zahl geändert werden. Die Zahl darf sich auf den meisten Systemen zwischen 1 und $65535=2^{16}-1$ bewegen. Allerdings muss das externe Postamt in Unix-Umgebungen bei einem Wert kleiner als 1024 mit root-Rechten gestartet werden.[30]

`epoLifeChecktime` ist eine Konstante vom Typ `Int` und gibt die Zeitspanne in Mikrosekunden an, in der Polls zur Prüfung der Systemintegrität von der Bibliothek durchgeführt werden. Die Zeitspanne gibt einmal an, in welchen Abständen, Readports ihre Existenz beim externen Postamt bestätigen müssen, bevor sie von diesem als nicht mehr existent angesehen werden, und die Zeitspanne in der der Link-Mechanismus (siehe Kapitel 5.4) überprüft, ob ein Readport, mit dem sich ein Thread verbunden hat, noch besteht.

Per default steht dieser Wert auf 30 Sekunden. Dies bedeutet, dass das externe Postamt den Ausfall eines Ports spätestens innerhalb einer Minute bemerkt. Um ein System zu erhalten, welches schneller auf Inkosistenzen reagiert, muss dieser Wert herabgesetzt werden. Um ein toleranteres System zu erhalten, muss er erhöht werden.

Das erste Kommunikationsprotokoll ist der Datentyp `EPOMessage`. Dieser beschreibt ein bidirektionales Protokoll zwischen einem portbasierten Prozesses und dem externen Postamt. Er beinhaltet folgende Anfragen:

`- EPOConnect PortDescriptor`

Diese Anfrage an das externe Postamt dient dazu, eine Leitung für den Port, der im `PortDescriptor` angegeben wurde, zur Verfügung zu stellen und diesen Port im externen Postamt anzumelden. Über die Leitung werden dann Daten, die an diesen Port gerichtet sind, dem portbasierten Prozess gesendet. Diese Anfrage schickt das interne Postamt an das externe, wenn ein Readport mittels `newPort` angelegt wird. Wird die Leitung, die zu dem Port gehört, geschlossen, so wird der Port automatisch im externen Postamt abgemeldet und aus seiner Datenbank ausgetragen.

`- EPOClose PortDescriptor`

Um einen Port beim externen Postamt abzumelden, sollte diese Nachricht an das externe Postamt geschickt werden. Das externe Postamt erkennt zwar mittels des eben beschriebenen LifeChecktime-Mechanismus selbst, ob ein Port noch existiert, doch ist dies die sauberere Lösung und meldet den Port sofort ab. Es kann schließlich sein, dass in einer sehr stabilen verteilten Umgebung der Wert von `epoLifeCheckTime` sehr hoch angesetzt wurde, so dass diese Abmeldeaufforderung sinnvoll ist.

`- EPOAccept`

Die erfolgreiche Bearbeitung einiger Anfragen (welcher, wird an entsprechender Stelle erwähnt) bestätigt das externe Postamt, indem es diese Nachricht an den portbasierten Prozess zurücksendet.

`- EPOReject`

Sendet das externe Postamt diese Nachricht an den portbasierten Prozess zurück, so heißt das, dass dessen Anfrage nicht erfolgreich bearbeitet werden konnte .

29 PortID ist nicht zu verwechseln mit PortDescriptor, der die Datenstruktur Port identifiziert. Es handelt sich um einen Datentypen aus dem ghc-Modul Socket (siehe [GHC] und S. 41),

30 Für genauere Informationen zu Sockets in Unix siehe [ASG].

- EPOSend PortDescriptor **Bool String**

Diese Meldung sendet ein portbasierter Prozess an ein externes Postamt, wenn innerhalb des Prozesses Daten via writePort auf einen außerhalb des Prozesses liegenden Port geschrieben werden, der bei diesem externen Postamt angemeldet ist. In dem String werden die Daten übergeben, die an den Port gesendet werden sollen. Ist der boolsche Parameter True, so sendet das externe Postamt beim erfolgreichem Empfang der Daten und Existenz des Ports EPOAccept. Bei Misserfolg sendet es dann EPOReject. Ist der Parameter False, so sendet das Postamt keine Antwort und ignoriert im Fehlerfall die Daten.

- EPORegister PortDescriptor PortName

Diese Nachricht wird von einem portbasierten Prozess dazu benutzt, einen mit EPOConnect angemeldeten Port, der durch den übergebenen PortDescriptor identifiziert wird, unter dem Namen in PortName zu registrieren, damit andere Prozesse diesen Port in Zukunft auch unter diesem Namen ansprechen können. Wenn das Registrieren funktioniert, schickt das externe Postamt EPOAccept zurück. Ist der Name bereits vergeben, oder der Port gar nicht angemeldet, schickt es EPOReject.

- EPOLookup PortName
- EPOPort PortDescriptor

Mit EPOLookup kann ein portbasierter Prozess beim externen Postamt den Portdescriptor eines Ports, der unter einem in PortName übergebenen Namen registriert ist, erfragen. Ist tatsächlich ein Port unter dem übergebenen Namen registriert, sendet das externe Postamt die Nachricht EPOPort mit dem entsprechenden PortDescriptor. Ist kein Port unter dem Namen registriert, sendet es EPOReject.

- EPOSendTo PortName **Bool String**

Die Nachricht EPOSendTo kann man als eine Kontraktion aus den Nachrichten EPOLookup und EPOSend ansehen. Mit ihr ist es möglich, Daten direkt an einen unter einem in PortName übergeben Namen im angesprochenen externen Postamt registrierten Port zu senden. Auch hier gibt der boolsche Parameter (wie bei EPOSend) an, ob das Postamt bei erfolgreicher Annahme der Daten eine Bestätigungsmeldung schicken soll. Bei Erfolg schickt es dann EPOAccept sonst EPOReject.

- EPOLiving ProcessID

Die EPOLiving-Nachricht, muss ein portbasierter Prozess alle epoLifeCheckTime Mikrosekunden an das externe Postamt senden, damit dieses seine Ports nicht abmeldet und den Prozess als tot deklariert. Der Prozess muss dem Postamt seine eigene Prozess-ID senden, die ja auch im PortDescriptor aller seiner angemeldeten Readports kodiert ist.

- EPOTest PortDescriptor

Mit dieser Nachricht kann ein portbasierter Prozess prüfen, ob ein bestimmter Port mit dem in PortDescriptor übergebenen Portdescriptor noch existiert. Wenn ja, sendet das Postamt EPOAccept, wenn nein EPOReject. Diese Nachricht wird von dem Link-Mechanismus, der in Kapitel 5.4 beschrieben wird, benutzt.

ii) Neue Socketzugriffsfunktionen (NSocket.hs)

Dieses Modul (NSocket, für den Quelltext siehe Anhang B, Bibliothekslisting 6) kapselt Zugriffe auf externe Verbindungen. Es verhindert, dass mehrere Threads gleichzeitig auf einen Handle zugreifen. Es funktioniert im Wesentlichen genauso wie das Modul Socket aus den GHC-Haskell Libraries ([HLi]), ist aber speziell auf die Anforderungen des externen Postamtes zugeschnitten. Das Modul wurde so angelegt, dass es später leicht mit einem Timeout-Mechanismus versehen werden kann.

Es definiert einen neuen Datentyp `NHandle`, der eine Synchronisation der Zugriffe auf ihn ermöglicht. Dieser besteht aus einer boolschen MVar einer MVar vom Typ `ThreadID` und einem normalen Handle, der in Haskell Input-Output-Streams beschreibt. Die boolsche MVar beschreibt ein Ticket für den Zugriff auf diesen Datentyp. Um auf diese Variable zuzugreifen, muss ein Thread den Wert aus dieser MVar nehmen (via `takeMVar`), sich also ein Ticket zum Zugriff besorgen. Ist die Variable leer, suspendiert somit der Thread, bis ein Wert vorliegt. Liest der Thread `True`, legt der Thread seine ID in der MVar vom Typ `ThreadID` ab und darf auf dem Handle operieren. Nach Abschluss der Operation muss der Thread den NHandle durch Zurückschreiben von `True` in die MVar wieder freigeben. Liest der Thread die Variable `False`, darf Thread nicht mehr auf dem Handle operieren und muss den Wert `False` wieder in die MVar zurücklegen. Ein Aufruf einer Funktion dieses Moduls mit einem NHandle, der an erster Stelle `False` enthält, löst eine Exception aus. `False` zeigt an, dass der Handle geschlossen ist.

Der Datentyp `NSocket` wird einfach direkt auf den Datentyp `Socket` des Moduls Socket der GHC-Haskell Libraries ([HLi]) abgebildet.

Einige der exportierten Funktionen dieses Moduls nutzen die intern in diesem Modul verwendete Funktion `waitOn`. Ihre Signatur ist:

```
waitOn :: IO t -> NHandle -> IO t
waitOn thread (ticket,threadmvar,handle) = ...
```

Sie führt eine in `thread` übergebene Funktion der IO-Monade mit Rückgabetyp `t` mit exklusivem Zugriff auf den `NHandle` als Thread aus und wartet darauf, dass er terminiert. Sie prüft ob `handle` offen ist, versucht ein positives Ticket (eines mit Inhalt `True`) zu aquirieren, startet die Funktion als Thread und speichert dessen Thread-ID in der `threadmvar` des `NHandles`. Um einen Timeout-Mechanismus hinzuzufügen, braucht dieser Funktion nur noch eine Wartefunktion mit einem anschließendem killThread hinzugefügt zu werden.

Das Modul exportiert folgende Funktionen:

`- nConnect :: PortHost -> IO NHandle`

Diese Funktion versucht eine Verbindung zum Socket (also zu `epoPortNumber`) eines externen Postamtes auf dem in PortHost übergebenen Rechners herzustellen und liefert diese als `NHandle` zurück. Sie entspricht der Funktion `connect` aus dem Modul Socket der GHC-Haskell-Libraries.

`- nListen :: IO NSocket`

`nListen` reserviert einen Socket, um auf ihm auf eingehende Verbindungen zu horchen. Sie liefert bei Erfolg den reservierten Socket in `NSocket` zurück. Diese Funktion entspricht `listen` aus dem Modul Socket der GHC-Haskell-Libraries.

`- nAccept :: NSocket -> IO NHandle`

`nAccept` nimmt Verbindungen auf einem mit `nListen` angelegten `NSocket` an und liefert den entsprechenden `NHandle` zurück. Die Funktion blockiert, bis eine Verbindung angenommen werden konnte. Auch hier kann bei Überschreiten eines Timeouts der Thread durch Benutzung der in NHandle abgelegten Thread-ID abgebrochen werden. `nAccept` entspricht `accept` aus dem Modul Socket der GHC-Haskell-Libraries.

`- nWrite :: NHandle -> String -> IO ()`

Mit `nWrite` können in String übergebene Daten auf einen `NHandle` geschrieben werden. Ist der `NHandle` geschlossen oder treten Probleme beim Senden auf, löst diese Funktion eine Exception aus. `nWrite` blockiert, bis die Daten gesendet wurden. Durch ein `threadKill` an die Thread-ID im `NHandle` lässt sich auch hier eine ewiges Blockieren der Funktion verhindern. `nWrite` entspricht `hPutStr` des Moduls IO der Haskell Libraries ([JHB99]).

- `nRead :: NHandle -> IO String`

`nRead` versucht von dem in `NHandle` übergebenen Handle Daten zu lesen und liefert diese zurück. `nRead` blockiert, bis Daten gelesen werden konnten. Ist der `NHandle` geschlossen oder treten Probleme beim Lesen auf, löst diese Funktion eine Exception aus. Ein ewiges Blockieren kann analog zu `nWrite` verhindert werden. `nRead` entspricht `hGetLine` des Moduls IO der Haskell Libraries ([JHB99]).

- `nClose :: NHandle -> IO ()`

Mit `nClose` wird der in `NHandle` übergebene Handle geschlossen. Dies entspricht der Funktion `hClose` des Moduls IO der Haskell Libraries ([JHB99]).

iii) Das eigentliche Prozessmodul (ExtPostOffice.hs)

In diesem Modul befindet sich die komplette Steuerung des externen Postamtes. Sein Quelltext ist in Anhang B, Bibliothekslisting 7 abgedruckt. Das externe Postamt ist im Wesentlichen eine kleine Datenbank. Es speichert Abbildungen von Portdescriptoren auf Verbindungen zu Ports, Namen auf Portdescriptoren und ihre Umkehrabbildung und verwaltet die zu einem Prozess gehörenden Ports (eine Abbildung von eines Prozesses auf eine Liste von Portdescriptoren).

Da alle diese Datenbanken von mehreren Threads bedient werden müssen, liegen sie alle in MVars. Um einen schnellen Zugriff auch auf viele Einträgen zu erreichen, werden die Datenbanken nicht in Listen sondern in FinateMaps (siehe für eine Erkärung von FinateMaps die GHC-Haskell-Libraries, [HLi]) geführt. Diese Datenbanken werden in den folgenden Strukturen abgelegt:

`Port2HandleMVar` bildet Portdescriptoren auf NHandles ab. Hier können also Verbindungen zu Readports nachgesehen werden.

`Name2PortMVar` bildet Portdescriptoren auf Namen ab. In dieser Datenbank kann also ein bereits registrierter Port nachgesehen werden.

`Port2NameMVar` ist die Umkehrabbildung von `Name2PortMVar`. Es werden also Portdescriptoren auf Namen abgebildet. Diese Datenbank wird benutzt, um beim Abmelden eines Ports auch seine Registrierung rückgängig zu machen.

`Process2Ports` speichert zu jedem Prozess die Liste der Readports, die er angelegt hat und einen Timestamp, wann der Prozess das letzte Mal seine Existenz bestätigt hat, um bei Bedarf seine allokierten Ports wieder abzumelden.

All diese Datenbanken werden in der Struktur `GlobalContext` zusammengefasst.

Die Hauptfunktion `main` dieses Moduls erzeugt die leeren Datenbanken, startet einen Thread aus der Funktion `lifeChecker`, um die verbundenen Prozesse zu überwachen, installiert einen Signalhandler, um Zugriffe auf dead-ended Socket-Verbindungen in Exceptions zu verwandeln,[31] und ruft eine Funktion `takeConnections` zur Annahme von Verbindungen auf.

Der Thread aus der Funktion `lifeChecker` prüft in Abständen von `epoLifeCheckTime` Mikrosekunden die Prozess-Port-Datenbank (`Process2Ports`), wann ein Prozess das letzte Mal seine Existenz bestätigt hat. Ist der letzte Timestamp eines Prozesses älter als 2 `epoLifeCheckTime` Mikrosekunden, wird der Prozess abgemeldet, und alle seine angemeldeten Ports sowie deren Registirierungen werden freigegeben.

31 Anmerkung für andere Haskellprogrammierer: Das lässt sich wunderbar kurz hinschreiben. Dennoch hat es fünfzig Stunden gekostet, den Fehler zu finden, der spontan das externe Postamt terminierte und die entsprechende Funktion in den Bibliotheken zu finden, die einen solchen Handler installiert. Erst durch den Versuch, das externe Postamt in C neu zu programmieren, um auch einen Debugger zur Verfügung zu haben, fiel die Idee auf den Befehl, wie solche Terminiersignale in C abgefangen werden. Man nutzt dafür einen Signalhandler. Dieser lässt sich auch im ghc mit einem Befehl des Posix-Modul der ghc-Haskell-Libraries ([HLi]) installieren.

Die Funktion `takeConnection` horcht auf einem Socket mit der Nummer `epoPortNumber`, der in `main` reserviert wurde, auf eingehende Verbindungen und startet zu deren Verarbeitung die Funktion `doCommunicate` als Thread.

`doCommunicate` liest die Anfrage auf der übergebenen Verbindung aus und wertet sie entsprechend des in `EPOMessage` definierten Protokolls aus. Für die Beschreibung dieses Protokolls siehe die Beschreibung von ExtPostOfficeConstants.hs.

Beim Übersetzen der Bibliothek wird aus diesem Modul ein eigenständiges Programm mit Namen ExtPostOffice erzeugt, was auf jedem Rechner, auf dem portbasierte Kommunikation eingesetzt werden soll, laufen muss.

iv) Zugriffsfunktionen auf externes Postamt (ExtPostOfficeAccess.hs)

In diesem Modul befinden sich die Funktionen, um das externe Postamt aus dem Modul Port.hs zu bedienen. Der Quelltext ist in Anhang B, Bibliothekslisting 4 abgedruckt. Das Modul ist ein Wrapper für das in `EPOMessage` definierte Protokoll zwischen portbasiertem Prozess und externem Postamt in Funktionen. Das Modul exportiert keine Datenstrukturen sondern nur folgende Funktionen:

- `epoHeartBeat :: IO ()`

`epoHeartBeat` ist eine Funktion, die vom internen Postamt (siehe dessen Beschreibung im folgenden Kapitel) als Thread gestartet wird. Sie sendet alle `epoCheckLifeTime` Mikrosekunden eine die Nachricht `EPOLiving` mit der eigenen Prozess-ID an das externe Postamt, um zu bestätigen, dass der aktuelle Prozess noch existiert.

- `epoTest :: PortDescriptor -> IO Bool`

`epoTest` wird vom Link-Mechanismus (Kapitel 5.4) dazu benutzt, um zu überprüfen, ob ein Readport mit dem in `PortDescriptor` übergebenen Portdescriptor noch existiert. Die Funktion sendet die Nachricht `EPOTest` mit dem zu überprüfenden Portdescriptor an das externe Postamt und wartet auf eine positive (`EPOAccept`) oder negative (`EPOReject`) Bestätigung. Existiert der Port noch, liefert die Funktion `True` sonst `False`.

- `epoConnect :: Read t => PortDescriptor -> Chan t -> IO`

Die Funktion `epoConnect` trägt einen mit `newPort` neu angelegten Readport mit dem in `PortDescriptor` übergebenen Portdescriptor im externen Postamt ein und startet im Falle einer positiven Rückmeldung (s.u.) einen Thread `epoConnectThread`, der Daten auf der mit dem externen Postamt etablierten Verbindung im Protokoll `EPOTransferMessage` empfängt und auf den in `Chan t` übergebenen getypten Channel des erstellten Readports schreibt. Der Typ `t` muss in der Klasse `Read` sein, da die aufgerufene Funktion `epoConnectThread` die Zeichenkettendaten, die auf der Verbindung zum externen Postamt empfangen werden, in den Typ `t` umwandeln muss, um sie auf den Channel vom Typ `t` schreiben zu können. `epoConnect` schickt die Nachricht `EPOConnect` mit dem Portdescriptor an das externe Postamt und wartet auf eine positive Bestätigunsmeldung (`EPOAccept`). Im Falle einer negativen Bestätigungsmeldung (`EPOReject`) löst die Funktion eine Exception aus.

- `epoRemove :: PortDescriptor -> IO ()`

`epoRemove` meldet einen Readport mit dem in `PortDescriptor` übergebenen Portdescriptor im externen Postamt wieder ab. Sie sendet `EPORemove` mit dem Portdescriptor an das externe Postamt. Auch diese Funktion wartet auf eine positive Bestätigunsmeldung (`EPOAccept`) und löst im Falle einer negativen Bestätigungsmeldung (`EPOReject`) eine Exception aus. Das externe Postamt sendet nach dem Erhalt dieser Nachricht auf der bei `epoConnect` etablierten Verbindung ein Terminiersignal, um den Thread, der dort Daten entgegen nimmt, zu beenden.

```
- epoRegister :: PortDescriptor -> PortName -> IO Bool
```

Diese Funktion versucht, einen Readport mit dem in PortDescriptor übergebenen Portdescriptor im externen Postamt unter dem in PortName übergebenen Namen zu registrieren. Sie sendet EPORegister mit dem Portdescriptor an das externe Postamt und wartet auf eine positive (EPOAccept) oder negative (EPOReject) Bestätigung. Bei einer positiven Bestätigungsmeldung bedeutet es, dass der Port unter dem übergebenen Namen registriert werden konnte und die Funktion liefert True. Bei einer negativen Bestätigungsmeldung heißt das, dass der Port bereits vorhanden ist und die Funktion liefert False zurück.

```
- epoLookup :: PortHost -> PortName
                        -> IO (Maybe PortDescriptor)
```

Die Funktion epoLookup sucht im externen Postamt des in PortHost übergebenen Rechners nach dem unter dem in PortName übergebenen Namen registrierten Port. Dazu wird die Nachricht EPOLookup mit dem zu suchenden Namen an das spezifizierte externe Postamt gesendet. Dieses sendet EPOPort mit dem Portdescriptor des gefundenen Ports zurück, falls es einen Port mit dem übergebenen Namen kennt. Wenn es keinen Port in dem angesprochenen Postamt gibt, der mit dem übergebenen Namen dort registriert wurde, sendet es EPOReject.

Wenn ein Portdescriptor pd zurückgeschickt wurde, liefert epoLookup als Ergebnis Just pd, sonst Nothing.

```
- epoSend :: PortDescriptor -> String -> Bool -> IO Bool
```

epoSend schickt die in String übergebenen Daten an das Postamt des Rechners, der in dem in PortDescriptor übergebenen Portdescriptor angegeben ist, damit es diese über die richtige Verbindung an den Port weiterleitet. Die Funktion sendet EPOSend mit dem Portdescriptor des Zielports, den Daten und einem boolschen Wert, der angibt, ob der Empfang der Daten vom externen Postamt quittiert werden soll. Wird True in dem boolschen Wert übergeben, wartet die Funktion auf eine positive (EPOAccept) oder negative (EPOReject) Bestätigungmeldung des angesprochenen Postamtes. Liefert das externe Postamt eine positive Meldung oder soll gar nicht auf eine solche gewartet werden, wenn also im boolschen Wert False übergeben wurde, ist das Ergebnis der Funktion True. Wird EPOReject empfangen, heißt das, dass der angesprochene Port nicht mehr existiert und die Funktion False zurückliefert.

```
- epoSendTo :: PortHost -> PortName -> String
                                    -> Bool -> IO Bool
```

Die Funktion epoSendTo ist analog zu der Funktion epoSend. Nur wird in ihr versucht, Daten an einen in PortName übergebenen Namen registrierten Port des externen Postamtes auf dem in PortHost übergebenen Rechner mit der Nachricht EPOSendTo zu senden. Die Rückgabewerte der Funktion werden genau wie bei epoSend gebildet.

c) Internes Postamt - Internal Post Office (IntPostOffice.hs)

Das interne Postamt (Internal Post Office, abgekürzt IPO) ist kein eigenständiger Prozess wie das externe Postamt sondern nur ein eigenständiger Thread, der in jedem portbasierten Prozess läuft. Er wird beim ersten Zugriff auf den Kommunikationschannel des internen Postamtes gestartet. Seinen Quelltext findet man in Anhang B, Bibliothekslisting 2.

Trotzdem ist sein Modul sehr ähnlich dem des externen Postamts aufgebaut. Da es weniger komplex ist und nur von einem Prozess aus bedient wird, befinden sich in diesem Modul sowohl das Kommunikationsprotokoll, der eigentliche Verwaltungsthread und die Zugriffsfunktionen auf diesen.

Das Protokoll zur Kommunikation mit dem Thread befindet sich im Datentyp `IPOMessage`. Der Channel, über den mit Hilfe dieses Protokolls kommuniziert wird, wird von der Funktion `ipoChannel` zurückgeliefert. Damit die Funktion `ipoChannel` einen Channel kreieren kann, muss sie aber eigentlich in der IO-Monade liegen, da auch die Funktion `newChan` aus Concurrent Haskell (siehe [JGF96] und [HLi]) in der IO-Monade liegt. Eine Funktion der IO-Monade wird aber bei jedem Aufruf neu ausgewertet, sonst könnte man z.B. nicht zweimal dasselbe hintereinander auf dem Bildschirm ausgeben. Da aber nicht bei jedem Zugriff auf `ipoChannel` ein neuer Channel erstellt werden soll, darf die Funktion nicht in der IO-Monade liegen. Dies erreicht man in Haskell mit dem Befehl `unsafePerformIO` aus dem Modul IOExts der ghc-Haskell Libraries ([HLi]). Dieser Befehl erlaubt einem den Aufruf von Funktionen der IO-Monade aus Funktionen, die nicht in der IO-Monade liegen, da er vom Typ `IO t -> t` ist. Die Verwendung von `unsafePerformIO` birgt allerdings Gefahren, da es damit möglich wird, Seiteneffekte in Funktionen auszulösen, die per Signatur keine haben dürften. Da `ipoChannel` aber immer aus einer Routine aufgerufen wird, die in der IO-Monade liegt und nicht von außerhalb der Bibliothek aufgerufen werden kann, ist der Einsatz in diesem Fall unbedenklich.

Der erste Zugriff auf `ipoChannel` startet automatisch den internen Thread des internen Postamtes (`ipoInternalThread`), die Funktion `epoHeartBeat` aus dem Modul extPostOfficeAccess als Thread und installiert den gleichen Signalhandler wie das externe Postamt.

Wie bereits in der Beschreibung zu PortGlobals angedeutet, sollten in diesem Modul noch zwei weitere Datentypen liegen, die aber zugunsten einer übersichtlicheren Modulstruktur dort notiert werden mussten. Es handelt sich um `IPOTransferMessage` und `IPOChannelType`. `IPOChannelType` ist der der Channel, über den bei der indirekten internen Kommunikation (siehe) die Daten als Zeichenketten an einen Port geschickt werden. Es handelt also nicht um einen reinen Stringchannel, da das Protokoll `IPOTransferMessage` genau wie beim externen Postamt `EPOTransferMessage` um eine Nachricht zum Terminieren der Verbindung erweitert ist. Die Aufforderung, Daten über den Channel zu senden heißt `IPOWrite`, die zum Terminieren `IPOTerminate`. `IPOChannelType` ist ein Channel vom Typ dieses Protokolls. In den meisten Fällen ist es für das Verständnis ausreichend, ihn einfach als Channel vom Typ `String` anzusehen. Deshalb werde ich auch im Normalfall hier von einem Stringchannel reden.

Auch das interne Postamt besitzt eine einfache Datenbank `IPOMap`. Sie bildet allerdings nur PortDescriptoren auf die eben beschriebenen Channels ab.

Das Protokoll in `IPOMessage` bietet folgende Kommunikationsmöglichkeiten:

`- IPOStore PortDescriptor IPOChannelType Qsem`

Mit `IPOStore` werden ein in PortDescriptor übergebener Portdescriptor und der in `IPOChannelType` übergebene zugehörige Stringchannel in die Datenbank des internen Threads eingetragen. In `QSem` wird eine Semaphore (für die Beschreibung einer Semaphore in Concurrent Haskell siehe Kapitel 3) mitgegeben, auf der nach dem Eintrag in die interne Datenbank ein Signal ausgelöst wird.

`- IPOLookup PortDescriptor (MVar (Maybe IPOChannelType))`

Die Nachricht `IPOLookup` versucht zu einem in `PortDescriptor` übergebenen Portdescriptor den zugehörigen Stringchannel nachzusehen. Wird er gefunden, wird er mit vorangestelltem `Just` in der mitgegebenen MVar zurückgeliefert, wenn nicht, wird `Nothing` zurückgeliefert.

- `IPORemove PortDescriptor QSem`

Mit `IPORemove` können der in `PortDescriptor` übergebene Portdescriptor und sein zugehöriger Channel aus der Datenbank gelöscht werden. Auf der in `QSem` übergebenen Semaphore wird nach Abschluss des Löschvorgangs ein Signal ausgelöst.

Der interne Thread `ipoInternalThread` des internen Postamts wertet Nachrichten auf dem `ipoChannel` aus und verändert die Datenbank entsprechend den eben beschriebenen Anweisungen.

Die Funktionen, die die Nachrichten an das interne Postamt schicken und von diesem Modul exportiert werden, sind `ipoConnect`, `ipoRemove` und `ipoLookup`:

```
- ipoConnect :: Read t => PortDescriptor -> IPOChannelType
                                         -> Chan t -> IO ()
```

`ipoConnect` trägt einen mit `newPort` neu angelegten Readport mit dem in `PortDescriptor` übergebenen Portdescriptor in der Datenbank des internen Postamt ein, indem es `ipoConnect` mit dem Portdescriptor und dem in `IPOChannelType` übergebenen Stringchannel über `ipoChannel` schickt, und startet einen Thread `ipoReceive`, der Daten auf dem in `IPOChannelType` übergebenen Stringchannel empfängt und auf den in `Chan t` übergebenen getypten Channel des erstellten Readports schreibt. Der Typ `t` muss in der Klasse `Read` sein, da die aufgerufene Funktion `ipoReceive` die Zeichenkettendaten, die auf der Verbindung zum externen Postamt empfangen werden, in den Typ `t` umwandeln muss, um sie auf den Channel vom Typ `t` schreiben zu können.

```
- ipoRemove :: PortDescriptor -> IO ()
```

Die Funktion `ipoRemove` entfernt einen Readport mit dem in `PortDescriptor` übergebenen Portdescriptor aus der Datenbank des internen Postamtes, indem sie `IPORemove` mit dem Portdescriptor über `ipoChannel` an das interne Postamt sendet. Existiert der Port nicht, so wird diese Anfrage ignoriert. Das interne Postamt sendet beim Empfang dieser Meldung über den entsprechenden Stringchannel das Terminiersignal an den Thread, der bei `ipoConnect` gestartet wurde.

```
- ipoLookup :: PortDescriptor -> IO (Maybe (IPOChannelType))
```

Die Funktion `ipoLookup` sucht in der Datenbank des internen Postamt nach dem Strinchannel des Ports des in `PortDescriptor` übergebenen Portdescriptors. Dazu wird die Nachricht `IPOLookup` mit dem zu suchenden Portdescriptor über `ipoChannel` gesendet. Dieses liefert sein Ergebnis über eine mitgeschickte MVar zurück.

Wenn der Channel `ch` gefunden wurde, liefert `ipoLookup` als Ergebnis `Just ch`, sonst `Nothing`.

d) Portschnittstelle (Port.hs)

Das Modul Port füllt all die Funktionen, die in Kapitel 5 (Anforderungsdefinition) beschrieben sind mit Leben. Es ist also die Implementation der Portschnittstelle. Es bedient das interne und das externe Postamt. Den kompletten Quelltext findet man in Anhang B, Bibliothekslisting 5.

In diesem Mosul wird die Datenstruktur `Port` definiert. Es ist das einzige Modul, welches die Konstruktoren `InternalPort` und `MergePort` kennt und benutzen kann.

Wie in Kapitel 6.2 bereits angedeutet, muss für Ports eine textuelle Darstellung existieren, damit sie über Netz verschickt werden können. Die Datenstruktur Port muss also in den Klassen `Read` und `Show` liegen. Da man von Ports des Typs `MergePort` (Ports mit Konstruktor `MergePort`) nur lesen und nur indirekt auf ihn schreiben kann, also nur der Thread sie benutzen kann, der auch die verschmolzenen Ports erstellt hat, existiert zu diesem Porttypen keine Darstellung. Ein Port mit Konstruktor `InternalPort` sollte über das Netzwerk verschickt werden können, allerdings darf man auf ihn nach der Dekodierung beim Empfänger

nur schreiben können. Da dieser Port über seinen PortDescriptor bestimmbar ist, wird nur dieser in seine textuellen Darstellung verwendet.

Beim Einlesen des versendeten Ports beim Empfänger lässt sich die Verwendung der Funktion `unsafePerformIO` leider nicht vermeiden, da der `PortInfo`-Anteil eines Ports mit Konstruktor `InternalPort` MVars aus Concurrent Haskell (siehe Kapitel 3) enthält, die nur in der IO-Monade erzeugt werden können, und ggf.. den Zugriff auf das interne Postamt erfordert, um bei einem prozessinternen Zielport den Stringchannel zu bestimmen. Da der empfangene Port in jedem Fall nicht lesbar ist, müssen keine Verwaltungsthreads gestartet werden. Das Problem besteht also nur darin die Datenstrukturen des `PortInfo`-Anteils anzulegen. Es werden auch nur prozessinterne Zugriffe durchgeführt. Das externe Postamt braucht hier nicht kontaktiert zu werden. Da es sich hier also um einen sehr klar eingegrenzten internen Zugriff handelt, ist der Einsatz von `unSafePerformIO` angebracht. So wird die textuelle Darstellung eines Ports vom Typen `InternalPort` einfach über die Darstellung seines Portdescriptors als Text realisiert. Das Einlesen gestaltet sich ein wenig komplexer, doch muss auch hier nur der Portdescriptor aus dem Text geparst werden und der `PortInfo`-Anteil ergänzt werden.

Um mit Ports noch ein wenig besser umgehen zu können, sie z. B. in sortierten Listen ablegen zu können, Ports vom Typen InternalPort noch in den Klassen `Eq` und `Ord`. Sie sind also vergleichbar und lassen sich total ordnen.

Neben der Datenstruktur Port werden im Anfangsteil dieses Moduls noch die Datentypen `MergePortMessage`, `PortInfo` und `PortChannel` definiert. `MergePortMessages` wird weiter unten im Zusammenhang mit der Funktion `mergePort` erklärt. `PortInfo` und `PortChannel` wurden bereits in Kapitel 6.2 besprochen.

Die Funktionen, die in diesem Modul definiert werden, sind in sechs Gruppen unterteilt:

1. Hilfsroutinen zur Portkonstruktion
2. Anlegen und Suchen von Ports
3. Registrierung von Ports
4. Verschmelzen von Ports
5. Zugriffe auf Ports
6. Zusatzthreadkontrollfunktionen

i) Hilfsroutinen zur Portkonstruktion

Hier werden die beiden Funktionen `refNrCounter` und `newRefNr` definiert.

`refNRCounter` definiert beim ersten Aufruf eine mit Null initialisierte MVar vom Typ Integer und liefert sie zurück. Bei den folgenden Aufrufen wird immer diese MVar zurückgegeben und nicht neu angelegt. Dies wird genauso wie bei der Funktion `ipoChannel` mit `unsafePerformIO` realisiert. Da auch diese Funktion nur von der Funktionen newRefNr, die in der IO-Monade liegt, aufgerufen wird, ist die Anwendung von `unsafePerformIO` auch hier unbedenklich.

`newRefNr` liefert bei jedem Aufruf eine 0, beim zweiten eine 1 und so weiter. Dafür greift sie auf die MVar aus `refNRCounter` zurück und inkrementiert sie bei jedem Aufruf.

ii) Anlegen und Suchen von Ports

Dieser Abschnitt des Moduls definiert die Funktionen `newPort`, `destroyPort`, `completePort` und die `Read`-Eigenschaft der Struktur `Port`. `newPort` wird vom Modul exportiert.

- `newPort :: (Show t, Read t) => IO (Port t)`

Die äußere Funktionsweise wurde bereits in Kapitel 5 festgelegt. Intern werden beim Anlegen eines neuen Readports folgende Schritte durchgeführt:

Für den Portdescriptor werden die IP-Adresse des aktuellen[32] Rechners, auf dem die Funktion aufgerufen wird, die ID des aktuellen Prozesses und über `newRefNr` eine eindeutige Referenznummer bestimmt. Für den `PortInfo`-Anteil wird ein getypter Channel und die aktuelle Thread-ID bestimmt, die in neu angelegten MVars abgelegt werden. Diese Daten werden anschließend in einem Port mit Konstruktor `InternalPort` zusammengesetzt. Mit `ipoConnect` und `epoConnect` werden die Daten an das interne und externe Postamt übermittelt und die Typumwandlungsthreads gestartet. Bevor der zusammengesetzte Port zurückgeliefert wird, wird an ihn ein sogenannter Finalizer[33] zur Zerstörung der in dem Port reservierten Datenstrukturen und angelegten Threads gebunden. Dieser Finalizer wird aufgerufen, wenn der Garbagecollector erkennt, dass der Port von keinem laufenden Thread mehr benutzt wird und freigegeben werden kann. In objektorientierten Programmiersprachen nennt man diese Finalizer Destruktoren. Der hier gebundene Finalizer ist die Funktion `destroyPort`.

- `destroyPort :: Port t -> IO ()`

Diese Funktion wird in newPort als Finalizer an einen Port gebunden und aufgerufen, wenn ein Port nicht mehr referenziert wird. Falls es sich um einen Port vom Typ InternalPort handelt, meldet sie den übergebenen Port mit `ipoRemove` und `epoRemove` beim externen Postamt ab und beendet dadurch auch die Typumwandlungsthreads. Im Fall eines Mergeports, wird über die Kommunikationsvariable die Terminieraufforderung an den `mergePortController` gesendet.

- `completePort :: PortDescriptor -> IO (Maybe (Port t))`

`completePort` ist eine Art interne Variante von `lookupPort`. Diese Funktion versucht zu einem PortDescriptor einen Port aufzubauen. Sie stellt zuerst fest, ob es sich um einen prozessinternen Port handelt, indem sie im Portdescriptor die die Prozess-ID und den Host überprüft. Wenn es sich tatsächlich um einen internen Port handelt, wird das interne Postamt mit `ipoLookup` kontaktiert, um den Stringchannel zu bekommen, der mit dem `PCipoChan`-Konstruktor in den Channel des `PortInfo`-Anteils eingetragen wird. Befindet sich der Port in einem anderen Prozess, wird in den `PortInfo`-Anteil eine externe Verbindung eingetragen und für die Thread-ID `Nothing`. Es wird an dieser Stelle also nicht überprüft, ob der Port tatsächlich existiert. Dies passiert erst beim ersten Schreibzugriff oder beim Start der Überwachungsfunktion. Wenn der Port gefunden wurde oder extern ist, liefert diese Funktion den Port vom Typ `InternalPort` mit einem vorangestelltem `Just` zurück. Liegt er intern und wurde nicht gefunden, liefert die Funktion `Nothing` zurück.

- `Read`

Die `Read`-Eigenschaft benutzt `completePort` zur Bestimmung des kompletten Ports und ruft dies innerhalb einer `unsafePerformIO`-Auswertung auf.

iii) Registrierung von Ports

Innerhalb dieses Abschnittes werden die beiden Funktionen `registerPort` und `lookupPort` definiert.

---

32 Aktuell heißt in diesem Zusammenhang immer, dass es sich um die Einheit handelt aus der eine bestimmte Funktion aufgerufen wurde. Z.B. um die IP-Nummer des Rechners, auf dem der Prozess läuft in dessen Thread gerade eine bestimmte Funktion zur Bestimmung dieser aktuellen IP-Nummer aufgerufen wird.

33 Für die genaue Beschreibung von Finalizern und Garbagecollecting-Mechanismen in Haskell siehe [JME].

- registerPort :: Port t -> PortName -> IO ()

registerPort nutzt die Funktion epoRegister um zu versuchen, den übergebenen Port unter dem übergebenen Namen im externen Postamt zu registrieren. Liefert epoRegister False zurück (die Registrierung schlug also fehl), löst diese Funktion eine Exception aus.

- lookupPort :: PortHost -> PortName -> IO (Port t)

lookupPort erwartet als Parameter einen Rechnernamen und den Namen eines registrierten Ports auf diesem Rechner. Sie versucht dann im externen Postamt des übergebenen Rechnernamens mit dem Befehl epoLookup und dem übergebenen Portnamen den Portdescriptor zu bestimmen und diesen dann mit completePort in einen Port vom Typ InternalPort zu ergänzen. Wird der Port im internen oder externen Postamt nicht gefunden, löst die Funktion eine Exception aus. Ansonsten liefert sie den Port als Ergebnis zurück.

- lookupLocalPort :: PortName -> IO (Port t)

Diese Funktion erwartet keinen Rechnernamen. Sie bestimmt ihn durch Anwendung der Funktion myIPAddress und ruft mit diesem Rechner und dem übergebenen Portnamen die Funktion lookupPort auf und liefert deren Ergebnis zurück.

iv) Verschmelzen von Ports

In diesem Abschnitt werden die Funktion mergePort, ihre Unterfunktionen und die Datenstruktur mergePortMessage definiert.

Die Unterfunktionen sind mergePortController und transferPort2Chan.

```
- mergePort :: (Show t1, Read t1, Show t2, Read t2) =>
                    Port t1 -> Port t2 -> IO (Port (Either t1 t2))
```

mergePort kann zwei beliebige Ports (auch MergePort) zu einem Port vom Typ MergePort verschmelzen. Wenn die Ausgangstypen der Ports t1 und t2 sind, so ist der Typ des entstehenden Ports (Either t1 t2). Die Funktion legt die MVars für die Kommunikationsvariable und die Rückgabevariable des Mergeports (siehe Definition MergePort im Quelltext oder in Listing 6.7) an, startet die Funktion mergePortController mit den beiden Ports und den beiden Variablen und der ID des aufrufenden Threads, setzt die Thread-ID der gestarteten Funktion und der beiden Variablen mit dem Konstruktor MergePort zusammen und liefert sie zurück.

```
- mergePortController:: (Show t1, Read t1, Show t2, Read t2) =>
                              Port t1 -> Port t2
                              -> MVar (Maybe (Either t1 t2))
                              -> MVar (MergePortMessage(Either t1 t2))
                              -> ThreadId -> IO ()
```

In mergePortMessage werden Anweisungen definiert, die an den mergePortController (der Thread, der mit der Funktion mergePortController gestartet wurde) geschickt werden können. Er reagiert auf folgende Anweisungen:

MPTerminate weist ihn an zu terminieren.

MPRead timeout veranlasst den mergePortController zwei Threads mit der Funktion transferPort2Chan zu starten, die ein Datum aus den beiden übergebenen Ports lesen und in einen Channel schreiben. Der eine Thread wird angewiesen, seinen Ergebnissen ein Left voranzustellen und der andere seinem Ergebnis ein Right voranzustellen. Auf diesen Channel wird mittels der Funktion readTimedChan, die in Abschnitt v definiert wird, suspendiert und auf einen Wert gewartet. Liegt innerhalb des von timeout angegeben Zeit kein Wert im Channel an, werden die beiden Threads angewiesen ihre Leseoperation zu unterbrechen, bzw. falls sie doch noch einen Wert lesen, diesen zurückzulegen. In diesem Fall wird in der Rückgabevariable Nothing zurückgegeben. Steht in timeout Nothing,

suspendiert der Controller, bis ein Wert anliegt. Wenn ein Wert gelesen werden konnte, wird er in der Rückgabevariable mit einem vorangestellten `Just` zurückgeben.

`MPUnread val` wird benutzt, um einen Wert `val` wieder in einen der beiden Teilports eines Mergeports zurückzulegen. Ist der Wert mit `Left subval`, wird die Funktion `unReadPort` (siehe auch Abschnitt v) mit dem ersten Teilport und `subval` aufgerufen, ist er `Right subval` wird `unReadPort` mit dem zweiten Teilport und `subval` aufgerufen.

v) Zugriffe auf Ports

In diesem Abschnitt werden die Funktionen `readTimedChan`, `writePort`, `writePortFast`, `readPort`, `readPortTimed` und `unreadPort` definiert. Die mittleren vier werden vom Modul Port exportiert.

`- readTimedChan :: Chan t -> Timeout -> IO t`

Dies ist eine Hilfsfunktion, um mit einem Timeout von einem Channel zu lesen. Wird als `Timeout Nothing` übergeben, ist diese Funktion identisch zu `readChan`. Die Timeout-Funktionalität wird realisiert, indem ein Thread gestartet wird, der von einem Channel versucht, einen Wert einzulesen und in eine MVar zu schreiben. Die Funktion sieht alle 100 Mikrosekunden nach, ob ein Wert in der MVar angekommen ist. Ist innerhalb der übergebenen Timeout-Zeit kein Wert angekommen, wird der Lesethread terminiert. Zur Sicherheit wird die MVar nach der Terminierung des Lesethreads noch einmal auf einen angekommenen Wert überprüft. Konnte ein Wert in der gewünschten Zeitspanne gelesen werden, wird dieser zurückgeliefert, ansonsten wird eine Exception ausgelöst.

`- writePort :: Show t => (Port t) -> t -> IO ()`

und

`- writePortFast :: Show t => (Port t) -> t -> IO ()`

Beide schreiben ein Datum auf einen Port. Die Funktionen betrachten den Typ des `PortChannel` im `PortInfo`-Anteil. Steht dort ein typisierter Channel (`PCTypedChan`), so wird bei beiden write-Funktionsaufrufen das Datum an diesen Channel via `writeChan` geschrieben. Steht dort ein Stringchannel (`PCipoChan`), wird das Datum mit `show` in eine Zeichenkette konvertiert und an diesen Channel geschickt. Auch dabei sind die write-Aufrufe identisch. Steht dort eine Verbindung zu einem externen Port, so wird `epoSend` benutzt und die Daten werden an den dort notierten Portdescriptor gesendet. Der boolsche Wert der `epoSend`- und `epoSendTo`-Nachricht wird je nach Aufruf von `writePort` und `writePortFast` gesetzt. Wird `writePort` aufgerufen und `epoSend` oder `epoSendTo` zeigen einen Fehler an, so löst die Funktion eine Exception aus.

An einen Port vom Typ `MergePort` können direkt keine Daten geschrieben werden.

`- sendToPort :: Show t => PortHost -> PortName -> t -> IO ()`

`- (<!!>) :: Show t => (PortHost,PortName) -> t -> IO ()`

`- sendToPortFast :: Show t => PortHost -> PortName -> t -> IO()`

Diese Funktionen senden die Nachricht `epoSendTo` mit dem übergebenen Portnamen und in `t` übergebenen Daten an das in PortHost übergebene externe Postamt. Die ersten beiden übergeben für den boolschen Wert in `epoSendTo True`, `sendToPortFast` übergibt `False`.

Wird nicht `sendToPortFast` aufgerufen und `epoSend` zeigt einen Fehler an, so lösen die Funktionen eine Exception aus.

`- sendToLocalPort :: Show t => PortName -> t -> IO ()`

`- sendToLocalPortFast :: Show t => PortName -> t -> IO ()`

Diese Funktionen arbeiten identisch zu den zuvor besprochenen, nur kontaktieren sie das externe Postamt des lokalen Rechners.

```
- readPortTimed :: (Read t, Show t) => Port t
                                         -> Timeout -> IO t
```

`readPortTimed` versucht innerhalb eines Timeouts einen Wert von einem Port zu lesen.Die Funktion überprüft als erstes, ob der aufrufende Thread berechtigt ist, von dem übergebenen Port zu lesen. Ist dies nicht gegeben, löst die Funktion eine Exception aus. Handelt es sich um einen Port vom Typ `InternalPort` liest die Funktion mittel `readTimedChan` einen Wert von dessen typisierten Channel. Handelt es sich um einen Port vom Typ `WritePort`, wird eine Leseaufforderung (`MPRead timeout`) über die Kommunikationsvariable an den entsprechenden `mergePortController` gesendet. Wenn in der Rückgabevariable `Nothing` geliefert wird, löst die Funktion eine Exception aus. Liegt dort ein Wert `Just val`, wird `val` zurückgeliefert.

```
- readPort :: (Read t, Show t) => Port t -> IO t
```

Diese Funktion ruft `readPortTimed` mit einem Timeout von Nothing auf, so dass `readPortTimed` suspendiert, bis ein Wert gelesen werden konnte.

```
- unreadPort :: Show t => Port t -> t -> IO ()
```

Diese Funktion legt gelesene Daten eines Readports oder Mergeports wieder zurück in dessen Channel. Sie kann nur von innerhalb des Moduls aufgerufen werden, weshalb keine Überprüfung der Zugriffsrechte durchgeführt wird. Handelt es sich um einen Readport, werden die Daten mittels `unGetChan` in den Channel des Ports zurückgelegt.[34] Handelt es sich um einen Mergeport, werden die zurückzulegenden Daten mittels der Nachricht `MPUnRead` über die Kommunikationsvariable an den `mergePortController` gesendet, der diese dann an den richtigen Port weiterleitet.

vi) Zusatzthreadkontrollfunktionen

Diese Funktionen implementieren den Link-Mechanismus. Der Datentyp `Link` wird über eine `ThreadID` definiert. Es ist die Thread-ID des Überwachungsthread.

In diesem Abschnitt werden die Funktion link mit seiner Unterfunktionen `processGuard` und `guard` und `unlink` definiert. `link`, `unlink` und der Datentyp `Link` werden vom Modul Port exportiert.

```
- link :: Port t -> IO Link
```

`link` richtet einen Überwachungsthread für den übergebenen Port ein. Die Funktion bestimmt die aktuelle Thread-ID und startet die Funktion `processGuard` mit dem übergeben Port und der aktuellen Thread-ID. Sie liefert die Thread-ID des gestarteten Überwachungsthreads zurück.

```
- processGuard :: Port t -> ThreadId -> IO ()
```

`processGuard` ruft `guard` mit dem zu dem übergebenen Port passenden Portdescriptor und der übergebenen Thread-ID auf. Wenn `processGuard` ein benannter Port übergeben wurde, wandelt er diesen mittel `lookupPort` in einen Port vom Typ `InternalPort` um und übergibt dessen Portdescriptor an `guard`. Wurde bereits ein Port vom Typ `InternalPort` übergeben, wird dessen Portdescriptor direkt an `guard` weitergereicht. Für Mergeports ist `processGuard` nicht definiert.

```
- guard :: PortDescriptor -> ThreadId -> IO ()
```

Die Funktion `guard` wartet immer `epoLifeCheckTime` Mikrosekunden und fragt anschließend mit `epoTest` und dem übergebenen Portdescriptor ein externes Postamt, ob der

34 Dies entspricht leider nicht den Tatsachen, da der Befehl unGetChan im ghc fehlerhaft ist. Deshalb kommt hier der Befehl writeChan zum Einsatz der damit die Reihenfolge der eingehenden Daten zerstört. Der andere Befehl steht bereits als Kommentar im Quelltext und sollte sobald dieser Fehler behoben ist ersetzt werden.

betreffende Port noch existiert. Ist dies nicht der Fall tötet `guard` den Thread, der in der `ThreadId` übergeben wurde, ansonsten beginnt die Funktion wieder von vorne.

- `unlink :: Link -> IO ()`

`unlink` tötet den in Link übergeben `guard`-Thread.

6.5 Sprachspezifische Probleme

In diesem Kapitel werden einige Fehler von Haskell dokumentiert. Dies ist für Leute interessant, die dieses Projekt erweitern möchten, um nicht noch einmal an einer Stelle Zeit zu verlieren, an der bereits zeit verloren wurde.

Da Haskell eine sehr junge Programmiersprache ist, treten bei der Entwicklung einer Bibliothek mit hohem Anteil an nebenläufigen Aktionen und Interaktionen mit Betriebssystemfunktionen für den ghc einige Probleme zu Tage.

Das Problem, welches eine Entwicklung unter dem ghc sicherlich am schwersten macht, ist ein fehlender Debugger. Man muss sich mit vielen `putStrLn`- oder `trace`-Befehlen behelfen.

Aber auch Fehler in den Bibliotheken und innerhalb der Dokumentation verhindern eine geradlinige Entwicklung.

So ist es leider manchmal nicht möglich einen `putStrLn`- oder `trace`-Befehl innerhalb von `unsafePerformIO` einzusetzen, das Programm verfängt sich dann in einem Deadlock. Dies macht das Debuggen bei der Suche nach Deadlocks noch etwas schwieriger.

Die Umsetzung der Socket-Bibliothek ist noch fehlerhaft. Bei nebenläufigem Zugriff auf ein und den selben Handle treten auch Deadlocks auf. Aus diesem Grunde entstand auch das NSocket-Modul.

Der Befehl `getCpuTime` liefert immer 10000000000.

Der Befehl `getHostEntry` liefert nicht das in der Dokumentation beschriebene Ergebnis bei der Bestimmung der IP-Adresse.

Um die Bibliothek modular zu gestalten, war es nötig zwei Interfacefiles der compilierten Bibliothek beizulegen, obwohl nur Funktionen und Schnittstellen eines Moduls nach außen zur Verfügung gestellt werden, was im Sinne der Datenkapselung als äußerst bedenklich einzustufen ist. Wenn ein Modul A ein anderes Modul B importiert und Funktionen oder Datentypen des Moduls benutzt um Datentypen aufzubauen, die es selbst abstrakt exportiert, so wird das Interfacefile des Moduls B benötigt, obwohl keiner der Datentypen von B an der Schnittstelle von A sichtbar ist.

7. Benutzung des Portkonzeptes

Dies ist die Anleitung zur Benutzung der von mir entwickelten Bibliotheken und Module.

Ich werde nach den Voraussetzungen, die man zur Benutzung der Bibliothek braucht, ein kurzes geführtes Beispiel angeben, an dem man den grundlegenden Umgang mit Ports kennen lernen kann.

7.1 Voraussetzungen

Damit Sie meine Bibliothek anwenden können, brauchen Sie den ghc 4.06 oder höher, gnumake und einen Texteditor (unter Unix bieten sich nedit oder emacs/xemacs an).

Ich gehe davon aus, dass Sie Grundkenntnisse im Umgang mit Haskell und dem ghc haben.

Stellen Sie sicher dass Sie das Programm mkdependHS (im Lieferumfang des ghcs) starten können. Ansonsten lokalisieren Sie es und kopieren Sie es in ein Verzeichnis, von dem aus Sie es starten können, oder nehmen Sie seinen Pfad mit in ihren Suchpfad auf.

Die Quelltexte zu meiner Diplomarbeit befinden sich auf der zu dieser Arbeit gehörenden CD. Kopieren Sie das Verzeichnis »portbasedCommunication« in irgendein Verzeichnis. Wechseln Sie in dieses Verzeichnis und starten Sie make. Dies sollte die Library und die Beispiele erzeugen.

7.2 Tutorium oder mein erstes portbasiertes Haskellprogramm

Nachdem Sie die Library erstellt haben, erstellen Sie an einem beliebigen Ort auf Ihrer Festplatte ein Verzeichnis, in welchem Sie Ihr erstes Haskell-Beispiel speichern wollen. Wechseln Sie in dieses Verzeichnis.

a) interne Kommunikation

Sie werden jetzt ein Beispiel erzeugen, um die interne Kommunikation mit Hilfe von Ports zu testen.

Legen Sie in dem von Ihnen gewählten Verzeichnis eine Datei mit dem Namen testinternal.hs an und bearbeiten Sie diese mit ihrem favorisierten Texteditor.

Schreiben Sie folgendes in die Datei:

```
import Port

main =
  do
    newport <- newPort
    forkIO (producer newport -1 0)
    forkIO (producer newport 1 1000)
    consumer newport

producer :: Port Integer -> Integer -> IO ()
```

```
producer p step counter =
  do
    p <!> counter
    producer p step (counter+step)

consumer :: Port Integer -> IO ()
consumer p =
  do
    val <- readPort p
    print val
    consumer p
```

Listing 7.1: testinternal.hs

Kopieren Sie das Makefile aus einem der Beispiele im »portbasedCommunication« in Ihr Verzeichnis.

Ändern Sie im Makefile die Variable PORTPATHauf das Verzeichnis in der die Datei libport.a liegt. Sie liegt nach Ausführen von make in »portbasedCommunication« dort unter port/lib.

Ändern sie SRCS auf testintern.hs, OBJS auf nichts (OBJS =) und PROGS auf testintern.

Starten Sie make. Starten Sie aus dem »portbasedCommunication/port/bin«-Verzeichnis ExtPostOffice. Dies ist das Serverprogramm, welches immer gestartet sein muss, wenn Sie auf einem Rechner portbasierte Prozesse laufen lassen möchten.

Anschließend können Sie testintern starten. In der Ausgabe sehen Sie durcheinander Zahlen einer von 0 aus aufsteigenden und Zahlen einer von 1000 aus absteigenden Folge.

Beenden Sie ihr Programm mit Strg-C oder schließen Sie einfach das Fenster in dem der Task läuft. Mit ein wenig Geduld werden Sie in der Ausgabe des externen Postamtes die Nachricht über die Zerstörung des angemeldeten Ports erhalten.

Die Funktion main legt mit `newport<-newPort` einen neuen Readport an. Anschließend werden mit `forkIO` zwei Producer-Threads gestartet. Der Port wird durch die Übergabe an einen anderen Thread zum Writeport.

Ein Provider gibt eine Zahlenfolge auf den Port von einem bestimmten Startwert aus mit einem festen Inkrement aus.

Der consumer gibt die Werte, die auf dem Port ankommen, auf dem Bildschirm aus.

b) externe Kommunikation

Um extern kommunizieren zu können, muss der Port aus testintern.hs registriert werden.

Fügen Sie deshalb hinter der Zeile `newport <- newPort` eine Zeile mit folgendem Inhalt ein:

```
    registerPort newport "test"
```

Legen Sie nun eine Datei testextern.hs mit folgendem Inhalt an:

```
import Port

main =
  do
    sendToLocalPort "test" 42
    main
```

Listing 7.2: testextern.hs

Fügen Sie im Makefile zu SRCS mit Blank getrennt testextern.hs und zu PROGS mit Blank getrennt testextern hinzu.

Starten Sie make. Starten Sie ExtPostOffice falls es nicht noch läuft. Anschließend testintern und zuletzt testextern. Sie sehen wieder die eine aufsteigende und die eine absteigende Zahlenfolge. Allerdings werden Sie aber sehr häufig auch die Zahl 42 entdecken.

Sie können testextern auch auf einem anderen Rechner ausführen. Allerdings müssen Sie dann für die Zeile

```
sendToLocalPort "test" 42
```

die Zeile

```
sendToPort "TestHost" "test" 42
```

schreiben.

Dabei muss *TestHost* der Rechner sein, auf dem testintern läuft. Vergessen Sie nicht, auf dem weiteren Rechner das externe Postamt zu starten.

7.3 Allgemeine Anleitung

Die Schnittstelle wurde bereits in Kapitel 5 ausführlich behandelt und auch genau so umgesetzt wie dort gefordert. Um die Funktionen in Ihrem Projekt einsetzen zu können, brauchen Sie Zugriff auf eine auf Ihrem System übersetzte portlib.a und die beiden Interface-Dateien Port.hi und PortGlobals.hi. Diese drei sollten sich im selben Verzeichnis befinden. Weiterhin sollte auf allen Rechnern, auf denen Sie dieses Projekt einsetzen möchten, das externe Postamt (ExtPostOffice) laufen.

Sollten die mitgelieferten Beispielmakefiles nicht ausreichen, um Ihr Projekt zu übersetzen, müssen Sie folgendes beachten:

Ein Programm, das portbasierte Kommunikation benutzen soll, muss mit der Library portlib.a gelinkt werden. Dafür müssen Sie ghc mit -L*Pfad* .-i*Pfad* den Pfad übergeben, in dem sich die Library und die Importdatein befinden und mit -lportlib.a den Namen der Library. Damit der ghc alle Sytembibliotheken findet, die für das Übersetzen eines portbasierten Projektes nötig sind, braucht er folgende Optionen:

-syslib concurrent -syslib net -syslib data -syslib util -syslib posix

Bedenken Sie, dass es in der Regel völlig ausreichend ist die Variablen PORTPATH, SRCS, OBJS, PROGS, EXTRA_LIBS, EXTRA_HC_OPTS im Beispiel zu verändern, um ein portbasiertes Programm (oder auch mehrere Programme in einem Projekt) zu erzeugen.

8. Beispiele

Anhand von zwei Beispielen soll hier ein Eindruck der Möglichkeiten vermittelt werden, die der Einsatz von portbasierten Haskellprogrammen bieten.

8.1 Datenbankserver und -client

Die Quelltexte dieses Beispiels finden Sie im »portbasedCommunication«-Verzeichnis unter examples/DB.

In vielen verteilten Anwendungen greifen Clients auf eine zentrale Datenbank zu. Dieses Beispiel zeigt, wie sich mit Hilfe von Ports solche Server und Clients realisieren lassen. Um das Beispiel zu compilieren tippen Sie make in diesem Verzeichnis (Achtung: Sie müssen vorher die Library und das externe Postamt übersetzt haben).

Um es unter Linux oder Unix zu starten, tippen Sie start. Das Script »start« startet auf einem Rechner das externe Postamt, den Datenbankserver und zwei Clients. Sie können dies selbstverständlich auch manuell tun (unter Windows ist dies ohnehin erforderlich, falls Sie dort keine bash installiert haben), allerdings müssen sie beachten, dass Sie zuerst das externe Postamt, dann den Datenbankserver und anschließend die Clients starten müssen.

a) Schnittstelle

Ein Datenbankserver muss verschiedene Anfrageaufforderungen bearbeiten können.

Wir wollen hier die Anfragen »Datensatz erzeugen«, »Datensatz nachsehen« und »Datensatz löschen« realisieren. Die Datenbank habe als Schlüssel eine ganze Zahl und als Wert eine Zeichenkette.

Diese Anfragen müssen auf einem registrierten Port gestellt werden. Die Antworten des Servers werden auf Ports, die bei der Anfrage verschickt wurden, zurückgesendet.

So ergibt sich folgende Schnittstellendatei Interface.hs (Listing 8.1), die auch dem Client bekannt sein muss.

Bei `Add` und `Del` liefert der Server bei Erfolg `True` und sonst `False` zurück. `Lookup` liefert, wenn der Schlüssel in der Datenbank vorhanden ist, den zugehörigen *Wert* als `Just` *`Wert`*.

```
{- Schnittstelle des Datenbankbeispiels -}

module Interface
  (
    DBMessages(..),
    ServerPort
  ) where

import IO
import Concurrent
import Port
import Maybe
import FiniteMap
```

```
data DBMessages =
    -- Eintrag zu Datenbank hinzufügen
    Add Integer String (Port Bool) |
    -- Eintrag in Datenbank nachsehen
    Lookup Integer (Port (Maybe String)) |
    -- Eintrag aus Datenbank löschen
    Del Integer (Port Bool)
      deriving (Show, Read)

type ServerPort = Port DBMessages
```

Listing 8.1: Interface.hs (Schnittstelle des Datenbankbeispiels)

`ServerPort` ist der Port, den der Datenbankserver registriert und anderen Prozessen für Ihre Anfragen zur Verfügung stellt.

b) Server

Der Server muss als erstes diesen Port vom Typ `ServerPort` mit `newPort` kreieren und ihn anschließend unter einem Namen, wir nehmen »DBServer«, registrieren (dies geschieht in der Funktion `main`).

Jetzt braucht er nur noch die Anfragen auf diesem Port entgegen zu nehmen (in der Routine `loop`), diese auszuwerten und entsprechende Antworten zu geben und die interne Datenbank zu führen (in `processMessages`). Den Quelltext finden Sie in Server.hs (Listing 8.2).

```
{- Der Server des Datenbankbeispiels -}

import IO
import Concurrent
import Port
import Maybe
import FiniteMap

import Interface -- Modul mit Protokollbeschreibung

{- die Datenbank: Integer-Schlüssel - String-Wert -}
type DB = FiniteMap Integer String

{- Funktion, die eingehende Anfragen bearbeitet. -}
processMessage :: DBMessages -> DB -> IO (DB)
processMessage message db =
    do
      putStrLn ("processMessage: Received" ++ (show message))
      case message of
        {- Anfrage: Neuen Wert str unter Schlüssel key in DB speichern. Erfolgsmeldung in
     checkport zurück -}
        (Add key str checkport) ->
          do
            let test = lookupFM db key
            -- Teste, ob vorhanden
            if (isNothing test)
              then
                do
                   checkport <!> True -- bestätige Empfang
                   putStrLn("processMessage: Updated DB")
                  return (addToFM db key str) -- Wert in DB eintragen
              else
                do
                   checkport <!> False -- Schon vorhanden
                   putStrLn("processMessage: Not updated DB")
                  return db -- DB unverändert zurück
```

```
    {- Anfrage: Eintrag mit Schlüssel key aus DB löschen. Erfolgsmeldung in checkport
       zurück -}
    (Del key checkport) ->
      do
        -- Teste, ob vorhanden
        let test = lookupFM db key
        if not (isNothing test)
          then
            do
              checkport <!> True -- bestätige Empfang
              putStrLn("processMessage: Deleted Entry")
              return (delFromFM db key) -- Eintrag in DB löschen
          else
            do
              checkport <!> False -- Schon vorhanden
              putStrLn("processMessage: Send complete:" ++
                       "Delete not possible!")
              return db -- DB unverändert zurück
    {- Anfrage: Wert mit Schlüssel key in DB nachsehen. Wert auf answerport
       zurückschicken -}
    (Lookup key answerport) ->
      do
        writePort answerport (lookupFM db key)
        putStrLn ("processMessage: Lookup done")
        return db -- DB unverändert zurück

main =
  do
    putStrLn "main: Opening new port.."
    {- Einen Port anlegen, auf dem Anfragen gestellt werden können -}
    serverport <- newPort
    putStrLn "main: Registering port.."
    {- Diesen Port registrieren -}
    registerPort (serverport :: ServerPort) "DBServer"
    putStrLn "main: Creating empty DB.."
    loop serverport emptyFM
  where
    loop :: ServerPort -> DB -> IO ()
    loop serverport db=
      do
        putStrLn "loop: Waiting for request.."
        message <- readPort serverport
        newdb <- processMessage message db
        loop serverport newdb
```

Listing 8.2: Server.hs (Server des Datenbankbeispiels)

c) Client

Ein Client sollte in der Lage sein, in Abhängigkeit von Benutzereingaben Datenbankanfragen an den Datenbankserver zu schicken und dessen Antworten auszugeben.

Der hier realisierte Client (Client.hs, Listing 8.3) wartet nach der Anzeige eines kleinen Menüs (`printMenu`) auf eine Benutzereingabe und erfragt entsprechend der Auswahl erforderliche Parameter vom Benutzer, sendet diese Daten anschließend als Anfrage an den Datenbankserver und stellt das Ergebnis der Anfrage lokal dar (`evalInput`).

```
{- Der Client des Datenbankbeispiels -}

import IO
import Concurrent
import Port
import Maybe
import FiniteMap
import Weak

import Interface -- Modul mit Protokollbeschreibung

{- Ein kurzes Menü mit möglichen Anfragen als Hilfestellung ausgeben -}
printMenu :: IO ()
printMenu =
  do
    putStrLn ""
    putStrLn "1) Neuen Datenbankeintrag"
    putStrLn "2) Eintrag in Datenbank nachsehen"
    putStrLn "3) Eintrag aus Datenbank löschen"
    putStrLn ""

{- Benutzereingabe lesen, Anfragen an Server schicken und Ausgabe anzeigen -}
evalInput :: ServerPort -> Port (Maybe String) -> Port Bool -> IO ()
evalInput messageport answerport checkport =
  do
    printMenu
    line <- getLine
    case line of
      "1" -> -- Neuen Datenbankeintrag generieren
        do
          putStr "Bitte Schlüsselwert eingeben (Zahl):"
          key <- readLn
          putStr "Bitte Wert eingeben (String):"
          str <- getLine
          messageport <!>(Add key str checkport) -- Anfrage senden
          check <- readPort checkport -- Antwort des Servers
          if check
            then
              do
                putStrLn "Der Wert wurde von der Datenbank akzeptiert"
            else
              do
                putStrLn ("Dieser Schlüssel scheint in der Datenbank"
                         ++ "bereits vorhanden zu sein.")
      "2" -> -- Eintrag in Datenbank nachsehen
        do
          putStr "Bitte Schlüsselwert eingeben (Zahl):"
          key <- readLn
          messageport <!>(Lookup key answerport) -- Anfrage senden
          answer <- readPort answerport -- Antwort des Servers
          putStrLn ("Die Datenbank sandte als Ergebnis:"
                   ++ (show answer))
```

```
      "3" -> -- Eintrag aus Datenbank löschen
        do
          putStr "Bitte Schlüsselwert eingeben (Zahl)"
          key <- readLn
          messageport <!>(Del key checkport) -- Anfrage senden
          check <- readPort checkport -- Antwort des Servers
          if check
            then
              do
                putStrLn "Der Wert wurde in der Datenbank gelöscht"
            else
              do
                putStrLn("Dieser Schlüssel scheint in der Datenbank"
                         ++ "nicht vorhanden zu sein")
      _ -> putStrLn "" -- falsche Eingabe ignorieren
    evalInput messageport answerport checkport

main =
  do
    {- Ports anlegen, auf denen die Antworten gesendet werden sollen -}
    putStrLn "main: Opening new ports (checkport, answerport) ..."
    checkport <- newPort
    answerport <- newPort
    {- Benutzereingaben auswerten -}
    p <- lookupLocalPort "DBServer"
    evalInput p answerport checkport
```

Listing 8.3: Client.hs (Client des Datenbankbeispiels)

8.2 Chatserver und -client

Dieses Beispiel demonstriert neben den Fähigkeiten der Client-Server-Kommunikation dieses Projektes eine Einsatzmöglichkeit des Befehls `mergePort` (siehe Client) und Absicherung der Produzenten gegen nicht mehr reagierende oder nicht mehr vorhandene Konsumenten (siehe Server).

Es handelt sich um ein System aus einem Chatserver und mehreren Chatclients.

Diese Clients können sich am Server anmelden und dort mit einem Benutzernamen authentifizieren. Sie erhalten ab diesem Zeitpunkt alle Nachrichten, die an den Server geschickt werden. Außerdem können die Clients selbst Nachrichten an den Server senden, die an alle angemeldeten Clients weitergesendet werden. So können sich auf diesem System mehrere Leute via Tastatureingaben gleichzeitig unterhalten.

Die Quelltexte dieses Beispiels finden Sie im »portbased haskell«-Verzeichnis unter examples/Chat. Um das Beispiel zu compilieren tippen Sie make in diesem Verzeichnis (Achtung: Sie müssen vorher die Library und das externe Postamt übersetzt haben).

Um es unter Linux oder Unix zu starten, tippen Sie start. Das Script »start« startet auf einem Rechner das externe Postamt, den Datenbankserver und zwei Clients. Sie können dies selbstverständlich auch manuell tun (unter Windows ist dies ohnehin erforderlich, falls Sie dort keine bash installiert haben), allerdings müssen sie beachten, dass Sie zuerst das externe Postamt, dann den Datenbankserver und anschließend die Clients starten müssen.

Probieren Sie auch einmal, Clients unsachgemäß zu beenden (ohne Eingabe einer Leerzeile), indem Sie das Fenster des Clients schließen oder ihn mit Strg-C abbrechen.

An den Ausgabe des Servers »`testWriteToPort: Dead port detected`« können Sie erkennen, dass er diesen Umstand, ohne selbst seine Funktionalität aufzugeben, erkennt und diesen Client abmeldet.

a) Schnittstelle

Von der Kommunikationsstruktur ist dieses Beispiel dem letzten sehr ähnlich. Auch hier besitzt der Server einen Port, auf dem in einem festgelegten Protokoll Anfragen gestellt werden können.

Dieses Protokoll (`Protocol`), der Typ des Anfrageports (`ServerPort`) und die Typen der Antwortports (`Message`) werden wieder in einem gemeinsamen Schnittstellenmodul implementiert. Dieses befindet sich in der Datei Interface.hs (Listing 8.4):

```
{- Schnittstelle des Chatbeispiels -}

module Interface
  (
    Protocol(..),
    ServerPort,
    Message
  ) where

import Port

-- Eine Nachricht besteht aus Benutzername und dem Nachrichtentext
type Message = (String,String)

data Protocol =
   -- Anmeldung mit dem Port, auf dem er Nachrichten empfangen möchte
   Connect (Port Protocol) |
   -- Abmelden mit dem Port, damit Zuordnung möglich
   Close (Port Protocol) |
   -- Daten senden hin (an Server) und zurück (zu den Clients)
   Send Message
   deriving (Show, Read)

-- Der Anfrageport
type ServerPort = Port Protocol
```

Listing 8.4: Interface.hs (Schnittstelle des Chatbeispiels)

b) Server

Der Aufbau des Servers dieses Beispiels ist nahezu äquivalent zu dem Aufbau des Datenbankservers. Auch hier muss der Server als erstes einen Port vom Typ `ServerPort` mit `newPort` kreieren, um Anfragen anzunehmen. Anschließend muss er diesen unter einem Namen, wir nehmen hier »ChatServer«, registrieren. Dies geschieht in der Routine `main`. In der Funktion `loop` nimmt er die Anfragen auf diesem Port entgegen, und wertet diese aus. Die Auswertung geschieht in der Funktion `processMessage`, in der auch die interne Portdatenbank entsprechend der Anfragen geupdated wird.

```
{- Ein Chatserver. An ihm können sich Chatclients anmelden und Nachrichten miteinander austauschen -}

import IO hiding (try)
import Maybe
import List
import Monad
import Port

import Interface -- Modul mit Protokollbeschreibung

{- die angemeldeten Empfangsports -}
type DB = [Port Protocol]
```

```
{- Anfrage auswerten und Portdatenbank dementsprechend ändern -}
processMessage :: Protocol -> DB -> IO DB
processMessage message db =
  do
    putStrLn ("processMessage: Received" ++ (show message))
    case message of
      (Connect messport) -> -- Messageport aufnehmen
        return (messport:db)
      (Close messport) -> -- Messageport entfernen
        return (delete messport db)
      (Send user_message) -> -- An alle die Nachricht senden
        do -- testWritePort auf alle ports anwenden und dabei
          -- die Datenbank neu aufbauen, damit nur noch funktionsfähige
          -- Ports in der Datenbank bleiben
          newdb <- foldM (\x y -> testWriteToPort user_message x y) [] db
          return newdb
  where
    -- Diese Funktion schreibt überwacht an einen Port
    -- Wenn das Schreiben funktioniert, wird er in die Liste aufgenommen, sonst nicht
    testWriteToPort :: Message -> DB -> Port Protocol -> IO DB
    testWriteToPort user_message portlist port =
      do
        test <- try (writePort port (Send user_message))
        case test of
          (Left _) -> -- es gab Probleme
            do
              putStrLn "testWriteToPort: Dead port detected"
              return portlist -- nicht aufnehmen
          (Right _) -> -- alles ok
            return (port:portlist) -- in Liste aufnehmen

main =
  do
    putStrLn "main: Opening new port..."
    -- Einen Anfrageport kreieren
    serverport <- newPort
    putStrLn "main: Registering port..."
    -- Port unter "ChatServer" registrieren
    registerPort (serverport :: ServerPort) "ChatServer"
    loop serverport []
  where
    -- Anfrage entgegen nehmen, auswerten und wieder von vorn
    loop :: ServerPort -> DB -> IO ()
    loop serverport db =
      do
        message <- readPort serverport
        newdb <- processMessage message db
        loop serverport newdb
```

Listing 8.5: Server.hs (Beispiel eines Chatservers)

Die wesentliche Neuerung zum Datenbankserver finden Sie in der Funktion `testWriteToPort`. Sie fängt einen eventuell fehlgeschlagenen Schreibzugriff auf einen nicht mehr existenten oder reagierenden Port mit Hilfe des `try`-Konstruktes ab.

So wird sichergestellt, dass die Nachrichten bei zukünftigen Zugriffen nicht mehr auf diesen Port geschrieben werden.

Den Quelltext finden Sie auch hier in Server.hs (Listing 8.5).

c) Client

Da der Client zwei Arten von Eingaben zu verwalten hat, bietet sich hier die Benutzung eines Mergeports an. Zum einen muss der Client die Nachrichten entgegen nehmen, die er vom

Chatserver geschickt bekommt und zum anderen die Nachrichten, die der Benutzer des Clients über Tastatur an die anderen Clients schicken möchte sowie die Terminieraufforderung[35].

Als erstes muss der Benutzer den Hostnamen angeben, auf dem der Chatserver läuft und sich mit einem frei gewählten Nutzernamen identifizieren. Daraufhin kreiert der Client mit `newPort` einen Port (`messageport`), auf dem er Nachrichten des Servers empfangen kann.

Mit diesem Port und dem Benutzernamen identifiziert sich der Client beim Chatserver (`chatserverport <!> (Connect messageport)`).

Nach dem Anmelden beim Chatserver kreiert der Client mit `newPort` einen weiteren Port (`keyboardport`), auf dem er Tastatureingaben empfangen kann und startet mittels `forkIO` einen Thread (`readKeyboard`), der dies bewerkstelligt und die Daten auf diesen Port schreibt. Diese beiden Readports werden mit `mergePort` (`<|>`) zu einem Mergeport (`port`) verschmolzen, der einer Auswertungsfunktion (`loop`) übergeben wird, die terminiert, wenn sie von der Tastatur eine leere Eingabe empfängt. Nicht leere Eingaben schickt die Auswertungsfunktion an den Chatserver und Nachrichten vom Chatserver gibt sie auf dem Bildschirm aus.

Den Quelltext finden Sie auch hier in Client.hs (Listing 8.6).

```
{- Der Client des Chatbeispiels -}

import IO
import Port

import Interface

main =
  do
    putStrLn "** Chatclient **"
    putStrLn "Leere Zeile: Programmende"
    putStrLn "Beschriebene: Nachricht senden"
    putStrLn ""
    putStr "Chathost (empty=local):"
    host <- getLine -- Host des Chatservers vom Benutzer erfragen
    chatserverport <- if (host=="") then lookupLocalPort "ChatServer"
                                    else lookupPort host "ChatServer"
    putStr "Benutzername: "
    user <- getLine -- Benutzernamen erfragen
    messageport <- newPort -- auf disem Port sollen Nachrichten empf. werden
    chatserverport <!> (Connect messageport) -- beim Server anmelden
    keyboardport <- newPort -- auf diesem Port Tastatureingaben empfangen
    forkIO (readKeyboard keyboardport) -- Tastaturleseprozess starten
    port <- keyboardport <|> messageport -- Verschmelze die beiden Ports
    loop port chatserverport user -- Werte Eingaben darauf aus
    chatserverport <!> (Close messageport) -- Ordentlich abmelden
  where
    -- Thread, der Eingaben von der Tastatur liest und auf einen Port schreibt
    readKeyboard :: Port String -> IO ()
    readKeyboard port =
      do
        line <- getLine
        port <!> line
        if not (line == "")
          then readKeyboard port
          else return ()
```

35 Da die Terminieraufforderung auch über Tastatur eingegeben wird, ist hier nur ein Merge über zwei Ports nötig: einer, der Netzwerknachrichten, und einer, der Tastatureingaben überträgt.

```
-- Auswertungsroutine, für Daten, die auf dem Mergeport ankommen
loop :: Port (Either String Protocol) -> Port Protocol
                                      -> String -> IO ()
loop inputport sendport user=
  do
    val <- readPort inputport-- Daten lesen
    case val of -- und auswerten
      (Left keyinput) -> -- Tastatureingabe
        if not (keyinput == "")
          then
            do
              sendport <!>(Send (user,keyinput))
              loop inputport sendport user
          else
            return () -- Ende
      (Right (Send (otheruser, message))) -> -- Nachricht von außen
        do
          putStrLn(otheruser ++ ": " ++ message)
          loop inputport sendport user
```

Listing 8.6: Client.hs (Beispiel eines Chatclients)

9. Zusammenfassung und Ausblick

Haskell ist eine junge, moderne Programmiersprache. Als funktionale Programmiersprache unterscheidet sie sich von traditionellen imperativen Programmiersprachen, bietet aber dennoch die Möglichkeit Ein- und Ausgabeoperationen durchzuführen. Im Gegensatz zu der funktionalen Programmiersprache Erlang ist Haskell getypt und nutzt eine call-by-need Auswertungsstrategie. Es existiert ein ausgereiftes Konzept zur nebenläufigen Programmierung mit einer nahtlosen Integration in Haskells Typkonzept (Concurrent Haskell). Zwar gibt es einige Ansätze zur verteilten Programmierung für Haskell, dennoch ist keiner dieser für heutige Anforderungen verteilter Systeme geeignet. Solche Anwendungen im verteilten Kontext -z.B. im Internet- erfordern eine robuste asynchrone Kommunikation mit der Möglichkeit, Verbindungen zwischen dynamisch gestarteten Prozessen zu etablieren. Vorbild für die Realisierung eines solchen Konzeptes, ist die Lösung, wie sie in Erlang bereits erfolgreich praktiziert wird. Jedoch ist die Berücksichtigung des Typkonzeptes von Haskell bei der Kommunikation wünschenswert.

Innerhalb dieser Arbeit wurde eine Bibliothek entwickelt, die in Haskell die Möglichkeit eröffnet, verteilte Systeme mit einem robusten bzw. dynamischen Charakter zu programmieren. Das bedeutet, dass Nachrichten zwischen Threads, die in unterschiedlichen Prozessen auf unterschiedlichen Rechnern laufen, ausgetauscht werden können. Diese Kommunikation ist robust und kann auch zwischen dynamisch gestarteten Prozessen aufgebaut werden. Die Bibliothek bietet einem die Möglichkeit, mit demselben Konzept sowohl nebenläufige als auch verteilte Systeme zu erstellen und zwischen ihnen zu migirieren. Das Typkonzept von Haskell bleibt dabei gewahrt, und seine Einhaltung wird bei Kommunikation über das Netzwerk zur Laufzeit überprüft. Interne Kommunikation wird auf die Kommunikation über Channels aus Concurrent Haskell abgebildet. Dadurch bleibt bei der internen Kommunikation Haskells call-by-need-Auswertungsstrategie erhalten. Bei der externen Kommunikation werden Ausdrücke vollständig reduziert, bevor sie verschickt werden.

Zur Realisierung des Konzeptes wurde ein portbasierter Ansatz verwendet. Die zentrale Datenstruktur, die bei dieser Kommunikation eingesetzt wird, ist ein Port. Es handelt sich um einen abstrakten Datentyp, wodurch eine interne Umstrukturierung der Implementation keine Auswirkung auf Anwendungen hat, die diese Bibliothek benutzen. Die wichtigste Eigenschaft eines solchen Ports ist, dass nur der Thread, der ihn erstellt hat, von ihm lesen darf, aber beliebig viele Threads auf ihn schreiben dürfen. Erst durch diesen Ansatz wird eine Implementierung robuster verteilter Systeme ermöglicht. Ports haben im Gegensatz zu der Mailbox in Erlang einen Datentyp und ein Thread kann mehrere Ports besitzen. Dadurch ist eine typisierte Kommunikation innerhalb eines verteilten Systems möglich. Dies erhöht die Sicherheit bei der Erstellung eines solchen Systems und seine Effizienz.

Die Bibliothek besteht in der Hauptsache aus drei Modulen und einem Verwaltungsprozess, dem externen Postamt. Dieser Prozess muss auf jedem Rechner, auf denen diese Bibliothek zum Einsatz kommen soll, genau einmal gestartet sein. Das externe Postamt kennt alle Ports, die auf dem Rechner, auf dem es läuft, kreiert wurden. Es reicht eingehende Daten für einen Port an den entsprechenden Prozess weiter. Das externe Postamt überwacht durch einen

Polling-Mechanismus die angemeldeten Ports. Eine weitere seiner Aufgaben ist die namentliche Registrierung von Ports, damit sie von anderen unabhängig gestarteten Prozessen identifiziert und angesprochen werden können. Zum externen Postamt gehört ein Modul, welches die Zugriffsfunktionen auf das externe Postamt enthält. Im Modul des internen Postamtes liegen Verwaltungsfunktionen für die prozessinterne Kommunikation. Das interne Postamt selbst ist ein eigenständiger Thread, der in jedem Prozess läuft. Und schließlich beinhaltet die Bibliothek das Modul Port, welches die Schnittstelle und den abstrakten Datentyp zur Verfügung stellt und das interne und externe Postamt bedient.

Die Bibliothek ist eine Erweiterung in Haskell für Haskell. D.h., dass die komplette Implementation in Haskell und mit Hilfe der Libraries des ghcs durchgeführt wurde. Es handelt sich somit um eine Erweiterung in Haskell mit den Mitteln die der Haskell-Compiler ghc zur Verfügung stellt. Auf den Einsatz eines Precompilers und Zugriffe auf programmiersprachenfremde Routinen wurde gänzlich verzichtet.

Zur Arbeit gehören zwei Client-Server-Beispielapplikationen. Diese geben einem die Möglichkeit, auch selbst Erfahrung mit diesem portbasierten Konzept zu sammeln. Zum einen wird eine Datenbankapplikation vorgestellt, die demonstriert, wie leicht portbasierte Anwendungen formuliert werden können. Das zweite Beispiel demonstriert die Tauglichkeit des Konzeptes für Internetanwendungen und seine Robustheit.

Die Implementation ist auf zukünftige Erweiterungen ausgelegt. Auf der einen Seite können durch den modularen Aufbau Veränderungen gezielt an einem Modul durchgeführt werden, ohne die Funktionalität anderer Module zu beeinflussen. Durch die strenge Kapselung der verwendeten Datentypen kann auch deren Umsetzung verändert werden, ohne dass Projekte, die mit dieser Bibliothek realisiert wurden, in ihrer Wirkungsweise beeinträchtigt werden.

Wenn das Laufzeittypsystem in Haskell auch auf Typen von Concurrent Haskell ausgeweitet worden ist, kann durch die Anwendung dynamischer Typen auf dieses Konzept, eine wesentliche Vereinfachung der Verwaltung des internen Postamtes und eine noch höhere Typsicherheit beim Senden von Daten über externe Verbindungen realisiert werden.

Die Effizienz der externen Datenübertragung kann durch Einführung eines binären, nicht mehr textbasierten, ggf. auch komprimierten Übertragungscode erhöht werden. Gleichzeitig steigt dabei auch die Sicherheit des Systems. Mit dem gleichen Aufwand können Authentifizierungs- und Verschlüsselungsmechanismen implementiert werden.

Um die Anzahl der benutzten Socketverbindungen zu reduzieren, ist eine Änderung des Socketmoduls des Projektes erforderlich. Eine Idee einer solchen Realisierung wäre die Einführung von Funktionen, welche mehrere Verbindung in einer zusammenfassen.

Lässt man die Registrierung von Ports unterschiedlicher Prozesse mit dem selben Namen auf dem gleichen Rechner zu, so steigert man die Flexibilität der Registrierung von Ports. Diese Änderungen werden nur in der Implementation des externen Postamtes und dem Modul seiner Zugriffsfunktionen wirksam. Um eine globale Registrierung und Suche von Ports zu etablieren, müssten ein oder mehrere übergeordnete externe Postämter eingeführt werden, die in der Lage sind, auch Ports anderer Rechner zu registrieren. Auch wäre es denkbar, jedem neu gestartetem Prozess auf einem Rechner einen auf diesem Rechner eindeutigen Namen zuzuweisen, so dass die Namen der registrierten Ports nur noch innerhalb dieses Prozesses eindeutig sein müssten.

Als relativ junge Sprache ist die Akzeptanz gegenüber Haskell noch gering. Diese Bibliothek erschließt Haskell völlig neue Anwendungsgebiete, insbesondere im Bereich Internet und kritischer verteilter Systeme. So trägt diese Arbeit dazu bei, die Attraktivität der Programmiersprache Haskell zu steigern.

Anhang A:
Benutzte Hard- und Software

Hardware

- PC mit Athlon-550 auf einem K7M
- 512MB RAM

Software

- Betriebssystem: Suse-Linux 6.4 ([SuS])
- Kernel: Suse-Kernel 2.2.14
- Simulationsumgebung: VMware ([VMw])
- Fensteroberfläche: KDE 1.1.2 ([KDE])
- Wordprocessor: StarOffice 5.1a und 5.2 ([SOf])
- Quelltexteditor: nedit ([Ned]) mit dem Syntaxhiglighting-Modul der Haskell-Homepage ([Has]).
- Compiler: ghc-4.06 ([GHC])

Auch auf diesem System sind bereits erhebliche Wartezeiten (im Bereich mehrerer Minuten) beim Compilieren in Kauf zu nehmen.

Anhang B:
Quelltexte der entwickelten Bibliothek

Diese Quelltexte findet man auch auf der beigefügten CD im Verzeichnis »portbasedCommunication«. Da es der Hauptauftrag dieser Diplomarbeit war, diese Bibliothek zu entwickeln, werden sie hier im Anhang abgedruckt. Im Gegensatz zu vielen anderen kommentiere ich meine Quelltexte, so dass sich vielleicht sogar ein Blick auf diese lohnt.

Ich hoffe, dass dieser Druck demjenigen, der meine Bibliothek um einige der in der Zusammenfassung vorgeschlagenen Neuerungen erweitern möchte, als Referenz dienen kann.

Bibliotheksbaustein 1: PortGlobals.hs

```
{- Typen und Funktionen für das Gesamte Portprojekt -}

module PortGlobals
  (
    PortDescriptor(..),
    PortHost,
    PortName,
    -- für nur zwei Interfacefiles
    IPOChannelType,
    IPOTransferMessage(..),
    -- Hilfsroutinen
    myIPAddress,
    nop,
    forget
  ) where

import IO
import Posix
import BSD -- für getHostName <- nicht Hugs-kompatibel! in syslib net
import SocketPrim hiding(sendTo) -- für inet_ntoa
import Concurrent

{- Das, was jetzt folgt, gehört eigentlich in IntPostOffice.hs. Ist aber wg. Einschränkungen im Modulkonzept
vom ghc nicht möglich. -}

-- Nachrichten, die über den internen Stringchannel gehen
data IPOTransferMessage=
   {- Daten, die verschickt werden und übersetzt werden sollen -}
   IPOWrite String |
   {- Aufforderung, den Thread, der zu dieser Verbindung gehört zu schließen -}
   IPOTerminate

-- Das wird den Portdescriptoren zugeordnet
type IPOChannelType = Chan IPOTransferMessage

{- ein paar Typen -}
type PortHost=String -- IP-Nummer als String
type PortName=String -- Name eines Ports
```

```
{- In PortDescriptor steht der Host des Ports, die ProzessID und eine inkrementell vergebene
Portreferenznummer -}
data PortDescriptor =
  PortDescriptor
    {
      pHost      :: PortHost,       -- IP-Addresse des Hosts als String
      pProcessId :: ProcessID,      -- Prozessnummer
      pRefNr     :: Integer         -- Eindeutige Referenznummer
    }
    deriving (Read, Show, Ord, Eq)

{------------------------------ Hilfsroutinen ----------------------------}
{- myIPAddress bestimmt die IP-Adresse des aktuellen Hosts als String -}
myIPAddress :: IO PortHost
myIPAddress =
  do
    {- Ip Adresse wird bei getHostEntry fehlerhaft bestimmt also Umweg über Hostname -}
    --hostentry <- getHostEntry -- Daten zur Ermittl. der IP-Addr. bestimmen
    myhostname <- getHostName
    hostentry <- getHostByName myhostname -- Daten zur Ermittl. der IP-Addr. bestimmen
    myhost <- (inet_ntoa (hostAddress hostentry)) -- IP-Adresse als String
    return myhost

{- tu nix! -}
nop :: IO ()
nop =
  do
    return ()
{- nimm Wert aber vergiss ihn, für übersichtlicheres Vergessen -}
forget :: a -> IO()
forget a = nop
```

Bibliothekslisting 1: PortGlobals.hs

Bibliotheksbaustein 2: IntPostOffice.hs

```
{-----------------------------------------------------------------------------
                              Das Interne Postamt und die Zugriffsfunktionen auf dieses
-----------------------------------------------------------------------------}
{- Die Abkürzung IPO bzw ipo steht für Internal Post Office -}


module IntPostOffice
  (
     ipoConnect,
     ipoLookup,
     ipoRemove,
    -- jetzt in PortGlobals    IPOChannelType,
     IPOTransferMessage(..)
  ) where

import IO
import IOExts
import Posix
import Concurrent
import Socket

import FiniteMap
import PortGlobals
import ExtPostOfficeAccess(epoHeartBeat)  -- wir brauchen nur das
```

```
{- in PortGlobals, damit nur zwei Interfacefiles benötigt werden
-- Nachrichten, die über den internen Stringchannel gehen
data IPOTransferMessage =
  -- Daten, die verschickt werden und übersetzt werden sollen
  IPOWrite String |
  -- Aufforderung, den Thread, der zu dieser Verbindung gehört, zu schließen
  IPOTerminate

-- Das wird den Portdescriptoren zugeordnet
type IPOChannelType = Chan IPOTransferMessage -}

{- Die Postamtkommunikationsstruktur -}
data IPOMessage =
   {- Nach Port Fragen und als Message zurückgeben LookupPort ist die Anweisung, einen Port auf der MVar
zu senden. Mit LookupPort kann man den Channel eines Ports im Postamt nachsehen -}
   IPOLookup
     {
       ipoLookupPortDesc:: PortDescriptor,
       ipoLookupMVar:: (MVar (Maybe IPOChannelType))
     } |
   {- StorePort ist die Anweisung, diesen Port im internen Postamt zu speichern. -}
   IPOStore
     {
       ipoStorePortDesc:: PortDescriptor,
       ipoStoreChan:: IPOChannelType,
       ipoStoreQSem:: QSem
     } |
   {- RemovePort ist die Anweisung, diesen Port zu löschen. Die mitgelieferte Semaphore kann man zur
Synchronisation verwenden -}
   IPORemove
     {
       ipoRemovePortDesc:: PortDescriptor,
       ipoRemoveQSem:: QSem
     } |
   {- Fehler mit Beschreibung -}
   IPOError
     {
       ipoError :: String
     }
{- Noch ein paar Hilfsroutinen für diese Kummunikationsstruktur -}
isError :: IPOMessage -> Bool
isError (IPOError _) = True
isError _ = False

{- Die Datenstruktur hält die Channels der zugehörigen Portdescriptoren. -}
type IPOMap = FiniteMap PortDescriptor IPOChannelType

{- Der Postofficekommunikationschannel. Hier wird auch gleichzeitig die zugehörige Auswertungsoutine
gestartet. Außerdem wird hier der heartbeat-Thread für das zur Benachrichtigung des externen Postamtest
gestartet. -}
ipoChannel :: Chan IPOMessage
ipoChannel =
  unsafePerformIO
  (
    do
      -- Zugriffe auf offene Pipes ignorieren
      installHandler sigPIPE Ignore Nothing
      ch <- newChan
      -- der channel darf nicht als Parameter übergeben werden, da der Prozess
      -- sonst nicht geforkt wird, warum ??? Antwort: Haskell!
      forkIO (ipoInternalThread emptyFM)
```

```
        -- Ein Prozess der dafür sorgt, dass die Readports im externen
        -- Postamt erhalten bleiben
        forkIO (epoHeartBeat)
        return ch
  )

{- Der Postoffice-Verwaltungsthread für interne Kommunikation. Dieser hält auch die FiniteMap für die
Zuordnung PortDesc - Channel -}
ipoInternalThread:: IPOMap -> IO ()
ipoInternalThread map=
  do
    -- Anfrage lesen
    request <- readChan ipoChannel
    -- debug -- putStrLn ("postOfficeInternalThread: request")
    -- Anfrage auswerten
    newmap <- evaluateRequest request map
    -- und von vorne
    ipoInternalThread newmap
  where
    evaluateRequest:: IPOMessage -> IPOMap -> IO IPOMap
    evaluateRequest request map=
      do
        case request of
          (IPOError str) -> error str
          -- Channel im internen Postamt nachsehen
          (IPOLookup pd chmvar) ->
            do
              -- debug -- putStrLn "postOfficeInternalThread: LookupPort"
              -- in Liste nachsehen
              putMVar chmvar (lookupFM map pd)
              return map -- Nachsehen verändert nicht
          -- Channel im internen Postamt speichern
          (IPOStore pd strch qsem) ->
            do
              -- putStrLn "postOfficeInternalThread: LoadPort" -- debug
              {- Es wird nicht überprüft, ob der Port schon im Postamt steht.
                    Dies sollte eigentlich auch nicht passieren, da nur meine Routinen
                    Zugriff auf das Postamt haben -}
              signalQSem qsem -- Wert ist jetzt gleich eingetragen
              -- debug -- putStrLn "postOfficeInternalThread: LoadPort Qsem reset"
              return (addToFM map pd strch)
          -- Port aus Postamt entfernen
          (IPORemove pd qsem) ->
            do
              -- Channel nachsehen
              case (lookupFM map pd) of
                (Just ch) ->
                  -- Terminieraufforderung auf Channel senden
                  writeChan ch IPOTerminate
                Nothing ->
                  nop -- Nicht gefunden, also nichts tun
              signalQSem qsem -- Aktion bestätigen, Wert ist gleich gelöscht
              -- debug -- putStrLn "postOfficeInternalThread: LoadPort Qsem reset"
              return (delFromFM map pd)
```

```
{-----------------------------------------------------------------------------
                                                  Schnittstellenfunktionen
-----------------------------------------------------------------------------}

{- Port im internen Postamt eintragen. Dabei wird der Empfangsthread der internen Daten gestartet -}
ipoConnect :: Read t => PortDescriptor -> IPOChannelType
                                      -> Chan t -> IO ()
ipoConnect pd stringchan typechan =
  do
    -- Bestätigungssemaphore
    sync <- newQSem 0
    -- Übergebe Stringchannel ans interne Postamt
    writeChan ipoChannel (IPOStore pd stringchan sync)
    -- Channel mit PCStringChannel verknüpfen (Übersetzerprozess)
    forkIO (ipoReceive stringchan typechan)
    -- Warte auf Empfangsquittierung
    waitQSem sync

{- Port aus interner Datenbank löschen. Ist er nicht vorhanden, passiert gar nichts -}
ipoRemove :: PortDescriptor -> IO ()
ipoRemove pd =
  do
    sync <- newQSem 0 -- Bestätigungssemaphore
    writeChan ipoChannel (IPORemove pd sync)
    waitQSem sync -- Bestätigung abwarten

{- Anfrage ans Postoffice nach einem Channel zu einem Portdescriptor -}
ipoLookup :: PortDescriptor -> IO (Maybe (IPOChannelType))
ipoLookup pd =
  do
    -- Empfangsvariable
    answermvar <- newEmptyMVar
    -- Übergebe Anfrageanweisung
    writeChan ipoChannel (IPOLookup pd answermvar)
    -- Variable auslesen (suspendiere, bis Ergebnis vorliegt)
    answer <- takeMVar answermvar
    -- Ergebnis zurückliefern
    return answer

{- Dieser Prozess wandelt einen Stringdatenstrom in getypte Daten. Ggf. wird über den inputchannel eine
Terminieraufforderung gesendet -}
ipoReceive :: Read t => IPOChannelType -> (Chan t) -> IO ()
ipoReceive inputchannel outputchannel =
  do
    -- putStrLn("ipoReceive: reading inputchannel") -- debug
    msg <- readChan inputchannel
    case msg of
      (IPOWrite str) -> -- Daten sollen übertragen werden
        do
          -- putStrLn("ipoReceive: read!" ++ str) -- debug
          writeChan outputchannel (read str)
          ipoReceive inputchannel outputchannel -- weitermachen
      IPOTerminate -> -- Terminierung gefordert
        nop
```

Bibliothekslisting 2: IntPostOffice.hs

Bibliotheksbaustein 3: ExtPostOfficeConstants.hs

```
{- Kommunikationsprotokoll und Portnummer des externen Servers und einige weitere Hilfsfunktionen
Autor: Ulrich Norbisrath
Erstellungsdatum: 27.04.2000
-}

module ExtPostOfficeConstants
  (
    epoPortNumber,
    epoLifeCheckTime,
    EPOMessage(..),
    EPOTransferMessage(..),
  ) where

-- import Concurrent
-- import IO
import Posix -- für getProcessID <- nicht Hugs-kompatibel! in syslib posix
import Socket -- für Portnumber

import PortGlobals

{- Eine Konstante für die Portnumber des externen Postoffice -}
epoPortNumber :: PortID
epoPortNumber = PortNumber 5600

{- lifeCheckTime ist eine Konstante, die die Mikrosekunden angibt, die eine Verbindung zu einem Readport
offen gehalten wird, ohne dass ihr Fortbestehen überprüft wird.
Eine Antwort auf eine Nachfrage muss innerhalb dieser Zeit zurückkommen -}

epoLifeCheckTime :: Int
epoLifeCheckTime = 30000000

{- Kommunikationsprotokoll mit dem externen Postoffice -}
data EPOMessage =
  {- Anforderung, eine Leitung für diesen Port zur Verfügung zu stellen, damit Daten an diesen gesendet
werden können. Diese Meldung schickt das interne Postamt an das externe, wenn ein Readport angelegt wird. Es
etabliert für diesen Port eine Leitung zum externen Postamt. Wenn nun Daten für diesen Port an das externe
Postamt geschickt werden, kann dieses die Daten auf der etablierten Leitung weiterschicken. Wird die Leitung,
die zu diesem Port gehört, geschlossen, so wird der Port aus der Liste des externen Postamtes ausgetragen. -}
  EPOConnect PortDescriptor |
  {- Um einen Port abzumelden, sollte man EPOClose benutzen. Diese Meldung wird nicht bestätigt. -}
  EPOClose PortDescriptor |
  {- Bestätigung, dass ein Auftrag bearbeitet wurde -}
  EPOAccept |
  {- Auftrag kann nicht ausgeführt werden -}
  EPOReject |
  {- Daten, die an einen Port gesendet werden sollen. Diese Aufforderung geht von einem Writer an das externe
Postamt eines Readports. Im zweiten Parameter kann angegeben werden, ob eine Bestätigungsmeldung
gewünscht wird, wenn die Daten angekommen sind. -}
  EPOSend PortDescriptor Bool String |
  {- Port unter Namen registrieren lassen, wird mit EPOAccept bzw. EPOReject bestätigt. Ein Port darf nur
einmal registriert werden, damit er auch wieder beim Abmelden gelöscht wird. -}
  EPORegister PortDescriptor PortName |
  {- Die Gegenfunktion zu EPORegister: sie sucht also nach einem zu einem Namen gehörenden
Portdescriptor. Sie schickt als Antwort entweder ein EPOReject oder über EPOPort den PortDescriptor -}
  EPOLookup PortName |
  {- erfolgreiche Antwort auf EpoLookup -}
  EPOPort PortDescriptor |
  {- EpoSendTo ist eine Kontraktion aus den Befehlen EPOLookup und EPOSend. Hiermit ist es möglich,
direkt an einen registrierten Port zu senden. Parameter: Name SendeBestätigung Daten. EPOPort wird also nur
von externen Postoffice gesendet -}
  EPOSendTo PortName Bool String |
```

```
    {- Mit den folgenden beiden Nachrichten wird überprüft, ob eine Verbindung noch besteht -}
    {- ExtLifeCheck | -- Lebst Du noch? wird seid 0.9 nicht mehr benötigt -}
    EPOLiving ProcessID|  -- Ja, ich lebe!
    {- Fragt nach, ob ein Port noch lebt. Wenn ja wird es durch EPOAccept bestätigt, sonst sendet das Postamt
EPOReject -}
    EPOTest PortDescriptor
    deriving (Read, Show, Eq)

{- Kommunikationsprotokoll über den Übertragunssocket -}
data EPOTransferMessage=
  {- Daten, die über eine mit EPOConnect angelegte Leitung geschrieben werden sollen -}
    EPOWrite String |
    {- Aufforderung, den Thread, der zu dieser Verbindung gehört zu schliessen -}
    EPOTerminate
    deriving (Read, Show, Eq)
```

Bibliothekslisting 3: ExtPostOfficeConstants.hs

Bibliotheksbaustein 4: ExtPostOfficeAccess.hs

```
{------------------------------------------------------------------------------
                          Zugriffsfunktionen auf das externe Postamt
------------------------------------------------------------------------------}
{- EPO bzw. epo sind Abkürzungen für External Post Office -}

module ExtPostOfficeAccess
  (
     epoHeartBeat,
     epoTest,
     epoRemove,
     epoConnect,
     epoRegister,
     epoLookup,
     epoSend,
     epoSendTo
  ) where

import IO
import Posix
import Socket
import SocketPrim hiding(sendTo)
import Concurrent

import PortGlobals
import ExtPostOfficeConstants

{- Diese Routine sendet alle checkLifeTime Mikrosekunden eine Bestätigung an das externe Postoffice, dass
dieser Prozess noch lebt -}
epoHeartBeat :: IO ()
epoHeartBeat =
  do
    -- Verbindung lokal aufbauen
    ipadr <- myIPAddress
    handle <- connectTo ipadr epoPortNumber
    -- Heartbeat generieren
    myprocessid <- getProcessID -- Prozess ID bestimmen
    hPutStrLn handle(show (EPOLiving myprocessid)) -- Lebensnachricht
    hClose handle
    threadDelay (epoLifeCheckTime) -- warten
    epoHeartBeat
```

```
{- Diese Routine testet, ob ein Port noch lebt. Wenn ja ist das Ergebnis true sonst false -}
epoTest :: PortDescriptor -> IO (Bool)
epoTest pd =
  do
    -- Verbindung lokal aufbauen
    ipadr <- myIPAddress
    handle <- connectTo (pHost pd) epoPortNumber
    -- Port übergeben
    hPutStrLn handle(show (EPOTest pd))
    -- Auf Bestätigung warten
    response <- hGetLine handle
    hClose handle
    case (read response) of
      EPOReject ->
        do -- Port existiert nicht mehr
          return False
      EPOAccept -> -- Port existiert
        do
          return True


{- Readport in externes Postamt eintragen, warten bis dieser Eintrag bestätigt wurde und den Handle dann
benutzen, um Daten aus dem externen Postamt zu empfangen. Diese Funktion startet den Thread für den
Readport, der Daten von extern empfängt und in den channel überträgt. Der Thread terminiert, wenn er über das
EPO die Nachricht bekommt. -}
epoConnect :: Read t => PortDescriptor -> Chan t -> IO ()
epoConnect pd ch =
  do
    -- Verbindung lokal aufbauen
    ipadr <- myIPAddress
    handle <- connectTo ipadr epoPortNumber
    -- Port übergeben
    hPutStrLn handle(show (EPOConnect pd))
    -- Auf Bestätigung warten
    response <- hGetLine handle
    case (read response) of
      EPOReject ->
        do -- Es gab Probleme, sollte nicht sein!
          hClose handle
          fail ("Can't connect port" ++ (show pd) ++
                " to external postoffice")
      EPOAccept ->
        do
          forkIO (epoConnectThread handle ch)
          nop -- Seltsam, dass forkIO nicht der letzte Befehl sein kann???
  where
    -- Handle muss noch beim Terminieren geschlossen werden
    epoConnectThread :: Read t => Handle -> Chan t -> IO ()
    epoConnectThread handle ch =
      do
        epoReceive handle ch
        hClose handle
```

```
{- ReadPort beim externen Postamt abmelden -}
epoRemove :: PortDescriptor -> IO ()
epoRemove pd =
  do
    -- Verbindung lokal aufbauen
    ipadr <- myIPAddress
    handle <- connectTo ipadr epoPortNumber
    -- Port übergeben
    hPutStrLn handle (show (EPOClose pd))
    -- Auf Bestätigung warten
    response <- hGetLine handle
    case (read response) of
      EPOReject ->
        do -- Es gab Probleme, sollte nicht sein!
          hClose handle
          fail ("Can't delete port" ++ (show pd) ++
                " from external postoffice.")
      EPOAccept ->
        do
          hClose handle

{- Diese Funktion überträgt Strings von einem Handle der beim externen Postamt angemeldet ist, an einen
getypten Channel. Diese Funktion wandelt einen Stringdatenstrom in einen Typdatenstrom. Zum Terminieren
muss über den Handle eine Terminieraufforderung geschickt werden. -}
epoReceive :: Read t => Handle -> Chan t -> IO ()
epoReceive handle ch =
  do
    line <- hGetLine handle -- diese Zeile blockierte in V0.8 warum???
    -- threadDelay 5000000 -- debug
    -- let line = "ExtLifeCheck" -- debug
    -- putStrLn ("epoReceive: read: " ++ line) -- debug
    case (read line) of
      (EPOWrite str) ->
        do
          writeChan ch (read str)
          epoReceive handle ch -- und weiter
      EPOTerminate -> -- Terminieraufforderung
        nop -- nix mehr tun
   {- Seit 0.9 unnötig (übernimmt epoHeartBeat)
    ExtLifeCheck -> -- Verbindungscheck
     do
       putStrLn "Received lifecheck request." -- debug
       hPutStrLn handle (show ExtLiving)
       putStrLn "Sent living answer" -- debug -}

{- Diese Funktion versucht einen Port im externen Postamt zu registrieren.  Bei Erfolg liefert sie True sonst False
-}
epoRegister :: PortDescriptor -> PortName -> IO Bool
epoRegister pdesc name =
  do
    -- Verbindung lokal aufbauen
    host <- myIPAddress
    handle <- connectTo host epoPortNumber
    -- Connectanfrage stellen
    hPutStrLn handle (show (EPORegister pdesc name))
    -- Auf Bestätigung warten
    response <- hGetLine handle
```

```
    case (read response) of
      EPOReject ->
        do
          hClose handle
          return False
      EPOAccept ->
        do
          hClose handle
          return True
```

{- Diese Funktion sucht in einem externen Postamt zu einem Namen einen Portdescriptor -}

```
epoLookup :: PortHost -> PortName -> IO (Maybe PortDescriptor)
epoLookup host name=
  do
    -- Verbindung aufbauen
    handle <- connectTo host epoPortNumber
    -- Connectanfrage stellen
    hPutStrLn handle(show (EPOLookup name))
    -- Auf Bestätigung warten
    response <- hGetLine handle
    hClose handle
    case (read response) of
      EPOReject ->
        do
          return Nothing
      (EPOPort pd) ->
        do
          return (Just pd)
```

{- epoSend schreibt Daten an einen Port an ein externes Postamt Sync gibt an, ob auf eine Bestätigung gewartet werden soll. Bei erfolgreichem Versand ist das Ergebnis True. -}

```
epoSend :: PortDescriptor -> String -> Bool -> IO Bool
epoSend pd str sync=
  do
    -- Verbindung aufbauen
    handle <- connectTo (pHost pd) epoPortNumber
    -- >Daten senden< schicken
    test <- try
      (hPutStrLn handle(show (EPOSend pd sync str)))
    case test of
      (Right _) -> -- kein Fehler beim Schreiben
        do
          if sync
            then -- Auf Bestätigung warten
              do
                response<- hGetLine handle
                case (read response) of
                  EPOReject->
                    do
                      hClose handle
                      return False
                  EPOAccept->
                    do -- alles in Ordnung
                      hClose handle
                      return True
            else return True -- keine Überprüfung gewünscht
      (Left exc) -> -- Exception aufgetreten
        do
          -- hClose handle, würde weitere Exception auslösen
          return False
```

```
{- epoSendTo schreibt Daten an einen Namen eines Ports an ein externes Postamt Sync gibt an, ob auf eine
Bestätigung gewartet werden soll. Bei erfolgreichem Versand ist das Ergebnis True. -}
epoSendTo :: PortHost -> PortName -> String -> Bool -> IO Bool
epoSendTo host name str sync =
  do
    -- Verbindung aufbauen
    handle <- connectTo host epoPortNumber
    -- >Daten senden< schicken
    test <- try
      (hPutStrLn handle (show (EPOSendTo name sync str)))
    case test of
      (Right _) -> -- kein Fehler beim Schreiben
        do
          if sync
            then -- Auf Bestätigung warten
              do
                response <- hGetLine handle
                case (read response) of
                  EPOReject ->
                    do
                      hClose handle
                      return False
                  EPOAccept ->
                    do -- alles in Ordnung
                      hClose handle
                      return True
            else return True -- keine Überprüfung gewünscht
      (Left exc) -> -- Exception aufgetreten
        do
          -- hClose handle, würde weitere Exception auslösen
          return False
```

Bibliothekslisting 4: ExtPostOfficeAccess.hs

Bibliotheksbaustein 5: Port.hs

```
{- Eine Datenstruktur für ein Multiwriter-, Singlereader-Portkonzept in Haskell
Autor: Ulrich Norbisrath
Erstellungsdatum: 14.03.2000
In diesem Modul wird die Datenstruktur Port definiert
-}

module Port
  (
    {- Datentypen, die exportiert werden -}
    Port, -- Port ist abstrakte Datenstruktur
    Link,
    Timeout,
    PortHost, PortName,
    {- Funktionen zum Registrieren, Finden und Anlegen von Ports -}
    newPort,
    registerPort,
    lookupPort,
    lookupLocalPort,
    {- Zugriffsfunktionen -}
    readPort,
    readPortTimed,
    writePort, (<!>),
    writePortFast,
    sendToPort, (<!!>),
    sendToPortFast,
    sendToLocalPort,
    sendToLocalPortFast,
```

```
    mergePort, (<|>),
    {- Zusatzthreadkontrollfunktionen -}
    link,
    unlink,
    {- Funktionen und Werte aus anderen Modulen -}
    ThreadId,
    forkIO,
    threadDelay,
    killThread,
    raiseInThread,
    yield,
    myThreadId,
    try -- entspricht tryAllIO aus Exception
  ) where

import Maybe
import IO hiding (try)
import Concurrent -- in syslib concurrent
import Channel
-- import FiniteMap -- in syslib data
import IOExts -- für unsafePerformIO in syslib lang
import Posix -- für getProcessID <- nicht Hugs-kompatibel! in syslib posix
import SocketPrim hiding (sendTo)
import Socket
import Weak -- um Ports aufzuräumen
-- import Dynamic funktioniert nicht für Channels, da diese nicht typable
import Exception hiding (try) -- für tryAllIO

-- eigene Module
import PortGlobals
import IntPostOffice
import ExtPostOfficeConstants
import ExtPostOfficeAccess

-- tryAllIO ist mächtiger
try=tryAllIO

{- Ein Port besteht aus seinem PortDescriptor (einer ID, die ihn eindeutig spezifiziert) und einem PortInfo-Anteil
(einige Zusatzinformationen wie Kommunikationschannel, ThreadID und Socketnummer)
Um Ports mergen zu können, kann er auch eine Konkettanation zweier Ports sein.
-}
data Port t =
  InternalPort -- Von diesen kann man manchmal lesen
  {
    pDesc :: PortDescriptor,
    pInfo :: PortInfo t
  } |
  MergePort -- Diese Ports sind ReadonlyPorts
  {
    mpThreadId :: ThreadId,
    mpMVar :: MVar (Maybe t), -- Maybe, da es vielleicht nicht im angeg. Timeout
                              -- geschafft wird
    mpMessage :: MVar (MergePortMessage t)
  }

data MergePortMessage t=
  MPTerminate | {- Terminierungsaufforderung -}
  MPRead Timeout | {- Leseaufforderung -}
  MPUnRead t {- Aufforderung, diesen Wert zurückzuschreiben -}

{- Funktionsweise von show für Port -}
instance Show (Port t) where
  show port =
    case port of
```

```
        -- Die Klammern  müssen sein, damit das Read immer eindeutig ist
        (InternalPort pdesc pinfo) -> "(" ++ (show pdesc) ++ ")"
        -- Mergeports sind nicht showbar und somit auch nicht verschickbar
        -- obsolet Namedports sind auch nicht showbar und nicht verschickbar

{- Funktionsweise von Gleichheit für Port. Sie sind gleich, wenn sie die selbe ID haben. Gleichheit ist nur für
gleiche Porttypen und nicht für Mergeports definiert. -}
instance Eq (Port t) where
  (InternalPort pd1 _) == (InternalPort pd2 _) = pd1       == pd2

{- Ordnung für Ports -}
instance Ord (Port t) where
  (InternalPort pd1 _) <  (InternalPort pd2 _) = pd1       <  pd2
  (InternalPort pd1 _) <= (InternalPort pd2 _) = pd1       <= pd2
  (InternalPort pd1 _) >  (InternalPort pd2 _) = pd1       >  pd2
  (InternalPort pd1 _) >= (InternalPort pd2 _) = pd1       >= pd2

{- In PortInfo stehen die Informationen des Ports, die für den lokalen Zugriff von Bedeutung sind, aber mit Hilfe
des Postamtes aus dem Portdescriptor generiert werden können. Zum einen ist das der Channel, über den Daten
geschrieben bzw- gelesen werden können (dieser ist entweder ein PCipoChannel, wenn es sich um einen
internen nicht sichtbaren Port handelt oder als Channel eines bel. Typ, wenn es ein interner Port ist oder als
Socket, wenn es ein externer Port ist) Weiterhin steht in der Datenstruktur die ThreadId des Threads, der den Port
generiert hat (bzw. das Recht von ihm zu lesen besitzt) oder wenn unbekannt Nothing. Über diese ID wird zur
Laufzeit entschieden, ob von diesem Port gelesen werden darf oder nicht. Diese ID ist eine MVar, damit eine
lesbare Kopie dieses Ports in einem anderen Thread erstellt werden kann. pTranslateId ist die ThreadId des
Threads, der Daten für den Ports von Strings in einen speziellen Typn übersetzt. pReceiveOrSendId ist die
ThreadID des Threads, der Daten für den Port aus einem Socket empfängt (bei InternalPorts) bzw. des Threads,
der die Portdaten an einen Socket sendet (bei Writeports). Die letzten beiden sind auch MVars, damit sich die
Ports mergen lassen.
 -}
data PortInfo t =
  PortInfo
    {
        -- Channel, über den Daten des Ports geschickt und gelesen werden
        pPortChan  :: MVar (PortChannel t),
        -- ThreadID des Threads, der von diesem Port lesen darf
        -- oder ThreadID des Thread für den dieser Port Readport ist
        -- (der ihn kreiert hat)
        pThreadId :: MVar (Maybe ThreadId)
        {- Diese Daten sind obsolet, das die Terminierung durch die ab V0.12 durch die Postämter durchgeführt
wird
        -- Stringchannel des Typübersetzerthreads, hierüber lässt
        -- sich auch eine Terminieraufforderung schicken
        pMessType :: MVar (Maybe IPOChannelType),
        -- ThreadID des Socket-Empfänger
        -- Besser wäre es, hier den Handle zu speichern, um zum Terminieren
        -- eine Nachricht zu schicken. Dies führt aber wg.
        -- eines Bugs im ghc zu Deadlocks
        pMessSocket :: MVar (Maybe ThreadId) -}
    }

{- Da bei externer Kommunikation über Strings bzw Sockets kommuniziert wird, ist dies ein spezieller Typ -}
data PortChannel t =   PCTypedChan (Chan t)
                     -- IPOChannelType:Stringchannel um Terminieraufforderung
                     -- erweitert
                     | PCipoChan (IPOChannelType)
                     -- hier gibt es ein wenig Redundanz zu Gunsten
                     -- der Übersichtlichkeit
                     | PCSocket PortDescriptor
                     | PCNothing -- Toter Port
```

```
{---------------------------------------------------------------------------
                              Hilfsroutinen zur Portkonstruktion
---------------------------------------------------------------------------}

{- Diese Funktion legt beim ersten Aufruf eine MVar an, danach liefert sie immer diese zurück -}
refNrCounter :: MVar Integer
refNrCounter =
  unsafePerformIO-- ohne unsafePerformIO wird das hier bei jedem Aufruf ausgeführt
    (
      do
        v <- newMVar 0
        return v
    )

{- Bei jedem Aufruf eine andere Nummer zurückgeben, klappt auch bei verteiltem Aufruf -}
newRefNr :: IO (Integer)
newRefNr =
  do
    -- mvar <- refNrCounter
    erg <- takeMVar refNrCounter
    putMVar refNrCounter(erg + 1)
    return erg

{---------------------------------------------------------------------------
                              Anlegen und Suchen von Ports
---------------------------------------------------------------------------}

{- newPort legt einen neuen Readport (Konstruktor: Port) für den aktuellen Thread an. Dabei wird der Port im
internen und externen Postamt eingetragen. Dieser Port ist dann insbesondere lesbar. Show t ist hier eigentlich als
Voraussetzung unnötig, aber erzwingt robustes Programmieren. Weiterhin werden beim Anmelden an das
interne und externe Postamt in dieser Funktion die beiden Transferthreads für die interne und externen
Typumwandlungen gestartet. -}
newPort :: (Show t, Read t) => IO (Port t)
newPort =
  do
    p <- createNewPort
    -- Destruktor zu Port hinzufügen
    addFinalizer p(destroyPort p) -- Finalizer an Port hängen
    return p
  where
    createNewPort=
      do
        -- Felder für Port füllen
        myhost      <- myIPAddress
        myprocessid<- getProcessID -- Prozess ID bestimmen
        newrefnr    <- newRefNr -- eindeutige neue Nummer holen
        typechan    <- (newChan :: IO (Chan t)) -- Hauptchannel
        portchan    <- newMVar (PCTypedChan typechan)
        mythreadid <- myThreadId
        pthreadid   <- newMVar (Just mythreadid)
        let pd = (PortDescriptor
                    myhost
                    myprocessid
                    newrefnr
                 ) -- end PortDescriptor
        ipochan <- (newChan :: IO (IPOChannelType)) -- Channel für int. PO
        -- Thread für interne (indirekte) Datenannahme starten
        ipoConnect pd ipochan typechan
        -- Thread für externe Datenannahme starten
        epoConnect pd typechan
```

```
        return
          (InternalPort
               pd
              (PortInfo
                 portchan
                 pthreadid
              ) -- end PortInfo
          ) -- end Port
```

{- destroyPort beendet alle mit dem Port verknüpften Prozesse, entfernt ihn aus der internen Datenbank, gibt ihn so dem Garbagecollector frei und löst bei einem Readport die Verbindung zum externen Postoffice. Kann nur modulintern ausgeführt werden -}

```
destroyPort :: Port t -> IO ()
destroyPort port =
  do
    putStrLn ("destroyPort: Destroying port" ++ (show port))
    case port of
      (InternalPort pdesc pinfo) ->
        doDestroyPort pdesc pinfo
      (MergePort mpthreadid pmmvar pmmessage) ->
        putMVar pmmessage MPTerminate -- Sende Terminieraufforderung
  where
    doDestroyPort :: PortDescriptor -> PortInfo t -> IO ()
    doDestroyPort pdesc pinfo =
      do
        portchan <- takeMVar (pPortChan pinfo)
        putMVar (pPortChan pinfo) PCNothing
        case portchan of
          PCNothing -> -- gar kein Channel mehr vorhanden, Port schon tot
            do
              nop
          _ ->
            do
              pthreadid <- takeMVar (pThreadId pinfo)
              ipoRemove pdesc -- intern abmelden
             -- ggf extern abmelden und Thread terminieren
              myprocessid <- getProcessID
              myhostname <- myIPAddress
             if ((isJust pthreadid) &&
                 ((pProcessId pdesc) == myprocessid) &&
                 ((pHost pdesc) == myhostname))
               then
                  epoRemove pdesc -- extern abmelden und Thread terminieren
               else
                  nop
             -- Port ungültig machen
              putMVar(pThreadId pinfo) Nothing
```

{- completePort sucht einen PortDescriptor im Postamt und liefert einen Port, auf den man schreiben kann, zurück. Als Parameter wird ein PortDescriptor übergeben. -}

```
completePort :: PortDescriptor -> IO (Maybe (Port t))
completePort portdesc =
  do
    -- Teste, ob nach internem Port gefragt ist
    myip <- myIPAddress
    mypid <- getProcessID
    if ((pHost portdesc) == myip) && ((pProcessId portdesc) == mypid)
      then -- interne Anfrage, aber Port nicht vorhanden
        do
          returnport <- completePortIntern portdesc
         return returnport
      else -- also extern
        do
          pthreadid <- newMVar Nothing
          pPortChan <- newMVar (PCSocket portdesc)
```

```
          return
            (Just
              (InternalPort
                portdesc
                (PortInfo
                  pPortChan
                  pthreadid
                ) -- end PortInfo
              ) -- end Port
            ) -- end Just
  where
    completePortIntern:: PortDescriptor -> IO (Maybe (Port t))
    completePortIntern portdesc=
      do
        strch <- ipoLookup portdesc
        if (isNothing strch)
          then
            do
              -- putStrLn "completePort: Nothing" -- debug
              return Nothing
          else
            do
              -- putStrLn "completePort: found" -- debug
              pthreadid<- newMVar Nothing
              pPortChan<- newMVar (PCipoChan (fromJust strch))
              return
                (Just
                  (InternalPort
                    portdesc
                    (PortInfo
                      pPortChan
                      pthreadid
                    ) -- end PortInfo
                  ) -- end Port
                ) -- end Just
```

```
{- Die Read-Eigenschaft des Ports wird über das Einlesen des Descriptors definiert und benutzt die Funktion
completePort für die eigentliche Porterstellung. Da ich nicht herausbekommen konnte, welche Readdarstellung
in Zukunft in Haskell verwendet wird, muss ich mit ifdefs arbeiten. -}
instance Read (Port t) where
{- Wenn diese trace-Anweisung benutzt wird, hängt das Programm. Es wäre ja auch zu einfach, wenn man
Haskell debuggen könnte.
  readsPrec n str = trace
            ("n: " ++ (show n) ++ " str: " ++ (show str))
            transferOld2Akt parsePorts -}
  readsPrec n str= transferOld2Akt parsePorts
    where
      -- Parse PortDescriptor
      parsePortDesc:: [(PortDescriptor,String)]
      parsePortDesc= transferAkt2Old(readsPrec n str)
      -- baue Liste von mögl. geparsten Ports auf (fast immer nur einer)
      parsePorts :: [(Port t,String)]
      parsePorts =
        foldl
          (\portlist pd -> portlist ++ (parsePort (fst pd) (snd pd)))
          [] parsePortDesc
      -- baue Port auf
      parsePort :: PortDescriptor -> String -> [(Port t,String)]
      parsePort pd str=
        case (unsafeLookupPort pd) of
          Nothing-> []
          (Just port) -> [(port,str)]
```

```
        -- Der vollständige Port, der hier unsafe gelesen wird, da wir in
        -- der Readfunktion sind
        unsafeLookupPort:: PortDescriptor -> Maybe (Port t)
        unsafeLookupPort pd= unsafePerformIO(completePort pd)
        -- wandle ggf. die Read-Darstellung in die alte um
        -- und wandle ggf. die alte Read-Darstellung in die neue zurück
#ifdef NEW_READS_REP
        -- wenn neue vorliegt
        transferAkt2Old:: Maybe (a,String) -> [(a,String)]
        transferAkt2Old new=
          if (isNothing new)
            then
              []
            else
              [fromJust new]
        transferOld2Akt:: [(a,String)] -> Maybe (a,String)
        transferOld2Akt old=
          if (length old) == 0
            then
              Nothing
            else
              Just (head old) -- or tail???
#else
        -- wenn alte vorliegt
        -- keine Umwandlung erforderlich
        transferAkt2Old akt= akt
        transferOld2Akt old= old
#endif

{-----------------------------------------------------------------------------
                                             Registrierung von Ports
-----------------------------------------------------------------------------}

{- RegisterPort versucht einen Port unter einem Namen im lokalen externen Postamt einzutragen. -}
registerPort :: Port t -> PortName -> IO ()
registerPort (InternalPort pd _) name =
  do
    test <- (epoRegister pd name)
    if test
      then
        nop
      else
        fail ("registerPort: Can't register port" ++
                        (show pd) ++ ".")

{- LookupPort sucht zu einem Host und einem Namen in einem externen Postamt nach einem Port und liefert
diesen, wenn vorhanden zurück. Ist er nicht vorhanden, wird eine Exception ausgelöst -}
lookupPort :: PortHost -> PortName -> IO (Port t)
lookupPort host name=
  do
    retval <- epoLookup host name
    case retval of
      Nothing -> fail ("Can't lookup port. Port not found".)
      (Just pd) ->
        do
          port <- completePort pd
          case port of
            Nothing ->
              do
                fail("Can't lookup port. Port not found".)
            (Just p) ->
              do
                return p
```

```
-- Lokaler lookup
lookupLocalPort :: PortName -> IO (Port t)
lookupLocalPort name=
  do
    myip <- myIPAddress
    port <- lookupPort myip name
    return port

{-------------------------------------------------------------------------
                                              Verschmelzen von Ports
-------------------------------------------------------------------------}

{- Zwei Ports zu einem neuen verschmelzen. -}
(<|>) :: (Show t1, Read t1, Show t2, Read t2) =>
               Port t1-> Port t2 -> IO (Port (Either t1 t2))
(<|>) = mergePort
mergePort :: (Show t1, Read t1, Show t2, Read t2) =>
               Port t1-> Port t2 -> IO (Port (Either t1 t2))
-- vier Möglichkeiten Ports zu verschmelzen
mergePort (InternalPort p11 p12) (InternalPort p21 p22) = -- 1
  doMergePort (InternalPort p11 p12) (InternalPort p21 p22)
mergePort (InternalPort p11 p12) (MergePort tid2 p21 p22) =    -- 2
  doMergePort (InternalPort p11 p12) (MergePort tid2 p21 p22)
mergePort (MergePort tid1 p11 p12) (InternalPort p21 p22) =    -- 3
  doMergePort (MergePort tid1 p11 p12) (InternalPort p21 p22)
mergePort (MergePort tid1 p11 p12) (MergePort tid2 p21 p22) =       -- 4
  doMergePort (MergePort tid1 p11 p12) (MergePort tid2 p21 p22)
doMergePort :: (Show t1, Read t1, Show t2, Read t2) =>
               Port t1-> Port t2 -> IO (Port (Either t1 t2))
doMergePort p1 p2=
  do
    mpmessage <- newEmptyMVar -- hier werden Nachrichten an den Port geschickt
    mpmvar <- newEmptyMVar -- hier soll der Wert zurückgegeben werden
    mythreadid <- myThreadId -- ThreadId des Controllers
    forkIO (mergePortController p1 p2 mpmvar mpmessage mythreadid)
    return (MergePort mythreadid mpmvar mpmessage)
  where
    {- Bei Bedarf ein Datum aus den Channels der Ports entgegen nehmen und in den neuen Channel
übertragen -}
    mergePortController:: (Show t1, Read t1, Show t2, Read t2) =>
                              Port t1-> Port t2
                                      -> MVar (Maybe (Either t1 t2))
                              -> MVar (MergePortMessage (Either t1 t2))
                              -> ThreadId -> IO ()
    mergePortController p1 p2 mpmvar mpmessage threadid=
      do
        message <- takeMVar mpmessage
        case message of
          (MPTerminate) -> return () -- Ende: Thread terminiert
          (MPRead timeout) -> -- Es soll ein Datum übertragen werden
            do
              allowed<- newMVar True -- Einmal Schreiben ist erlaubt
              backchan<- newChan -- Rückgabechannel (Chan, damit Timeoutfkt.
                                 -- angewendet werden kann, sonst wäre
                                 -- MVar ausreichend)
              thr1<- forkIO (transferPort2Chan p1 backchan timeout
                                 Left threadid allowed)
              thr2<- forkIO (transferPort2Chan p2 backchan timeout
                                 Right threadid allowed)
              test<- try (readTimedChan backchan timeout)
```

```
                case test of
                  (Right val) -> -- keine Exception, kein Timeout aufgetreten
                     putMVar mpmvar (Just val) -- Wert liefern
                  (Left exc) ->
                    do
                       test <- takeMVar allowed
                       putMVar allowed False -- weiteres Lesen verhindern
                      if test
                        then -- Es steht immer noch True in allowed
                             -- -> keiner gelesen
                           putMVar mpmvar Nothing -- "keinen" Wert liefern
                        else -- Es konnte doch noch einer lesen
                          do
                            -- Dann Wert auslesen (es muß jetzt einer vorh. sein)
                             val <- readChan backchan
                             putMVar mpmvar (Just val) -- Wert liefern
            (MPUnRead val) -> -- Wert an Mergeport zurückgeben
              do
                 putStrLn "mergePortController: Unreadrequest"
                case val of
                  (Left val1) -> -- Wert vom Typ t1
                     unReadPort p1 val1
                  (Right val2) -> -- Wert vom Typ t2
                     unReadPort p2 val2
          -- und von vorne
          mergePortController p1 p2 mpmvar mpmessage threadid
```

{- Wert aus Port bei Erlaubnis in Channel übertragen. Ggf. wieder in Port zurücklegen. Einer der beiden Threads wird auf jeden Fall den Wert zurücklegen müssen -}

```
transferPort2Chan :: (Show valuetype, Read valuetype) =>
                       Port valuetype -> Chan eithertype -> Timeout
                     -> (valuetype -> eithertype) -> ThreadId
                     -> MVar Bool -> IO ()
transferPort2Chan port backchan timeout transferfunc
                                        threadid allowed =
  do
    -- Auf jeden Fall Wert holen
    val <- readPortWithThreadId port timeout threadid
    test <- takeMVar allowed -- Schreiberlaubnis holen
    if test
      then -- dieser Thread darf schreiben
         writeChan backchan (transferfunc val)
      else -- dieser Thread darf nicht schreiben
         unReadPort port val -- dann Wert zurück
    putMVar allowed False -- Zeige: "Ich habe fertig"
```

{---
Zugriffe auf die Ports
---}

{- Teste, ob der Port dem aufrufenden Thread gehört, also, ob davon gelesen werden kann. Damit die Routine selbst nicht auf die ThreadMVar in Port zugreifen muss, wird deren Wert mit übergeben damit auch der zu überprüfende Thread festgelegt werden kann, wird diese Thread id auch mitgegeben -}

```
isReadPort :: Port t -> Maybe ThreadId -> ThreadId -> IO Bool
isReadPort port pthreadid mythreadid =
  do
    case port of
      (InternalPort pdesc pinfo) ->
        do
            myprocessid <- getProcessID
            myhostname <- myIPAddress
           return ((isJust pthreadid) &&
                ((fromJust pthreadid) == mythreadid) &&
                ((pProcessId pdesc) == myprocessid) &&
                ((pHost pdesc) == myhostname))
```

```
      (MergePort _ _ _) ->
        do
           return ((isJust pthreadid)
                          && ((fromJust pthreadid) == mythreadid))
```

{- Der Schreibzugriff ist sehr einfach, da das Schreiben auf Ports immer erlaubt ist. -}

```
writePort :: Show t => (Port t) -> t -> IO ()
writePort port val= writePortMsgSync port val True
(<!>) :: Show t => (Port t) -> t -> IO ()
(<!>) = writePort
writePortFast :: Show t => (Port t) -> t -> IO ()
writePortFast port val= writePortMsgSync port val False
```

{- Schreibzugriff an einen registrierten Port -}

```
sendToPort :: Show t => PortHost -> PortName -> t -> IO ()
sendToPort host name val= sendToPortMsgSync host name val True
sendToPortFast :: Show t => PortHost -> PortName -> t -> IO ()
sendToPortFast host name val= sendToPortMsgSync host name val False
(<!!>) :: Show t => (PortHost,PortName) -> t -> IO ()
(<!!>) (host,name) val = sendToPortMsgSync host name val True
sendToLocalPort :: Show t => PortName -> t -> IO ()
sendToLocalPort name val=
  do
    host <- myIPAddress
    sendToPortMsgSync host name val True
sendToLocalPortFast :: Show t => PortName -> t -> IO ()
sendToLocalPortFast name val=
  do
    host <- myIPAddress
    sendToPortMsgSync host name val False
```

{- Schreibzugriff an einen registrierten Port mit evt. Bestätigung -}

```
sendToPortMsgSync :: Show t => PortHost -> PortName -> t -> Bool -> IO ()
sendToPortMsgSync host name val sync=
  do -- Dann direkt an den Namen senden
    test <- epoSendTo host name(show val) sync
    if test
      then
        nop -- alles ok
      else
        fail ("writePort: Can't lookup port"++name++" on "++host)
```

{- Schreibzugriff auf einen Port, der ggf auf Bestätigung des Empfangs vom externen Postamt wartet -}

```
writePortMsgSync :: Show t => Port t -> t -> Bool -> IO ()
writePortMsgSync (MergePort _ _ _) _ _ =
  fail ("No write operation to MergePorts implemented (yet")).
writePortMsgSync (InternalPort pdesc pinfo) val sync =
  do
    portchan <- takeMVar (pPortChan pinfo)
    case portchan of
      (PCNothing) ->
        do
          putMVar(pPortChan pinfo) portchan
          fail ("writePort: port " ++ (show pdesc) ++ "is dead.")
      (PCTypedChan ch) ->
        do
          writeChan ch val
          putMVar(pPortChan pinfo) portchan
      (PCipoChan ch) ->
        do
          writeChan ch(IPOWrite (show val))
          putMVar(pPortChan pinfo) portchan
```

```
      (PCSocket pd) ->
        do
          test <- epoSend pd (show val) sync
          if test
            then -- Senden war ok
              putMVar(pPortChan pinfo) portchan
            else -- Senden war nicht ok
              do
                putMVar(pPortChan pinfo) portchan
                destroyPort(InternalPort pdesc pinfo)
                fail("writePort: port " ++ (show pd)
                                     ++"seems to be dead!")
```

{- Hilfsroutine, um mit einem timeout von einem Channel zu lesen. Der Timeout wird in ungefähren Mikrosekunden angegeben -}

```
timeoutFailure = "readTimedChan: Timeout reached, no value read".
type Timeout = Maybe Integer
readTimedChan :: Chan t -> Timeout -> IO t
readTimedChan ch timeout=
  do
    case timeout of
      Nothing ->
        do
          retval <- readChan ch
          return retval
      (Just timeout) ->
        do
          receiveval<- newEmptyMVar
          readerthread<- forkIO (channelReader ch receiveval)
          wait4Reader readerthread receiveval timeout
          -- Teste, ob Ergebnis vorliegt
          test <- isEmptyMVar receiveval
          if (test)
            then
              fail timeoutFailure
            else
              do
                retval<- takeMVar receiveval
                return retval
  where
    wait4Reader :: ThreadId -> MVar t -> Integer -> IO ()
    wait4Reader readerthread receiveval timeout=
      do
        if (timeout <= 0)
          then -- Timeout erreicht
            do
              killThread readerthread
              return ()
          else
            do
              test<- isEmptyMVar receiveval
              if (test)
                then -- noch kein Ergebnis
                  do
                    -- warte ein bischen
                    threadDelay 100
                    wait4Reader readerthread receiveval(timeout-100)
                    return ()
                else -- Ergebnis vorhanden
                  do
                    return ()
    channelReader :: Chan t -> MVar t -> IO ()
    channelReader ch receiveval=
      do
        retval <- readChan ch
        putMVar receiveval retval
```

```
{- Der Lesezugriff auf einen Port ist etwas komplexer, da nur der kreierende Thread auf diesen Port lesend
zugreifen darf. Da diese Operation ggf. blockiert, wird ein Parameter als Timeout (Anzahl mikrosekunden)
mitgegeben -}
readPort :: (Read t, Show t) => Port t -> IO t
readPort port = readPortTimed port Nothing

readPortTimed :: (Read t, Show t) => Port t -> Timeout -> IO t
readPortTimed port timeout =
  do
    -- putStrLn "readPort: just called." -- debug
    mythreadid <- myThreadId
    val <- readPortWithThreadId port timeout mythreadid
    return val

{- nur für den internen Gebrauch -}
readPortWithThreadId :: (Read t, Show t) => Port t -> Timeout
                                                -> ThreadId -> IO t
readPortWithThreadId port timeout mythreadid =
  do
    case port of
      (InternalPort pdesc pinfo) ->
         do
           pthreadid <- readMVar (pThreadId pinfo)
           test <- isReadPort port pthreadid mythreadid
           if test
             then
               do
                 pPortChan <- readMVar (pPortChan pinfo)
                 case pPortChan of
                   (PCTypedChan ch) ->
                     do
                       -- putStrLn "readPort: Left reading" -- debug
                        retval <- readTimedChan ch timeout
                       -- putStrLn ("readPort: Left read: " ++ (show retval)) -- debug
                       return retval
                 {- (PCipoChan ch) -> darf nicht vorkommen
                     do
                       -- putStrLn "readPort: Right reading" -- debug
                       retval <- readTimedChan ch timeout
                       -- putStrLn ("readPort: Right read: " ++ retval) -- debug
                       return (read retval)
                   (PCSocket _) -> -- darf nicht vorkommen
                     do
                       fail "readPort: Unknown error" -}
             else
               fail ("readPort: Thread is not allowed to read the port"
                     ++ (show  port) )
      (MergePort mpthreadid mpmvar mpmessage) ->
        do
          test <- isReadPort port (Just mpthreadid) mythreadid
          if test
            then
              do
                -- Leseaufforderung an Mergeportverwaltungsthread schicken
                 putMVar mpmessage (MPRead timeout)
                 retval <- takeMVar mpmvar -- timeout hier nicht nötig, da
                                           -- dies von dem Thread geregelt wird
                case retval of
                   Nothing -> -- Timeout ist aufgetreten
                     fail ("readPort: Timeout")
                  (Just val) ->
                    return val
```

```
            else
               fail("readPort: Thread is not allowed to" ++
                                    "read the mergeport")

{- ggf. das Lesen von einem Port rückgängig machen -}
unReadPort :: Show t => Port t -> t -> IO ()
unReadPort port val =
  do
    -- putStrLn "unReadPort: Unreadrequest"
    case port of
      (InternalPort pdesc pinfo) ->
        do -- Daten in Channel zurücklegen
          -- putStrLn ("unReadPort: for port "++(show pdesc)++
          --             " val "++(show val))
          portchan<- takeMVar (pPortChan pinfo)
          -- putStrLn ("unReadPort: doing soon")
          case portchan of
            -- tuts nicht: Bug in H., blockiert bzw hängt manchmal bei schnellem Hinter-
            -- einanderaufrufen
            -- (PCTypedChan ch) -> unGetChan ch val
            (PCTypedChan ch) -> writeChan ch val-- hinten anhängen
            {- darf nicht vorkommen
               (PCipoChan ch) -> unGetChan ch (show val) -}
          -- putStrLn ("unReadPort: nearlly done")
          putMVar(pPortChan pinfo) portchan
          -- putStrLn ("unReadPort: done")
      (MergePort tid mpmvar mpmessage) ->
        do
          -- Nachricht für unRead an MergePortController schicken
          putMVar mpmessage(MPUnRead val)

{---------------------------------------------------------------------------
                         Zusatzthreadkontrollfunktionen
  Man kann dadurch den Status eines Writeports überwachen lassen und im Falle des nicht mehr Funktionierens
                   den entsprechenden Thread automatisch terminieren lassen.
---------------------------------------------------------------------------}
type Link = ThreadId

{- link überwacht für einen Thread einen Port. Wenn sich auf ihn nicht mehr schreiben lässt, wird der zugehörige
Thread geschlossen. Diese Funktion nutzt die Fähigkeit, dass dies ohnehin regelmäßig vom externen Postamt
überprüft wird und führt einfach ständig ein Lookup dieses Ports durch. Wenn er nicht mehr existiert, wird der
Thread der diesen Link angelegt hat getötet. Funktioniert allerdings nicht für Mergeports (logisch, da readonly)
Es wird (noch) nicht überprüft, ob ein Port mehrfach vom selben Thread gelinkt wird -}
link :: Port t -> IO (Link)
link p =
  do
    mythreadid <- myThreadId -- ID für diesen Thread auslesen
    threadid <- forkIO (processGuard p mythreadid)
    return threadid
  where
    processGuard:: Port t -> ThreadId -> IO ()
    processGuard(InternalPort pdesc pinfo) thread =
      do
        guard pdesc thread
    processGuard(MergePort _ _ _) _ =
      do
        fail ("Can't link MergePort (yet)")
```

```
    guard :: PortDescriptor -> ThreadId -> IO ()
    guard pdesc thread=
      do
        threadDelay epoLifeCheckTime -- schlafe erstmal
        test <- epoTest pdesc -- Teste, ob Port noch vorhanden
        if test
          then -- Alles ok
            guard pdesc thread -- weiter
          else -- Port existiert nicht mehr -> zugehörigen Thread töten
            killThread thread

{- Stellt die Überwachung wieder ein -}
unlink :: Link -> IO ()
unlink l =
  do
    killThread l
```

Bibliothekslisting 5: Port.hs

Bibliotheksbaustein 6: NSocket.hs

```
{-----------------------------------------------------------------------------
                    Zugriffsfunktionen auf Sockets
    Sie verhindern, dass mehrfach gleichzeitig auf denselben Socket zugegriffen wird.
-----------------------------------------------------------------------------}

module NSocket
  (
    NHandle,
    NSocket,
    nRead,
    nWrite,
    nClose,
    nConnect,
    nListen,
    nAccept
  ) where

import IO
import Posix
import Socket
import SocketPrim hiding (sendTo,accept)
import Concurrent
import Exception

import PortGlobals
import ExtPostOfficeConstants

{- Der neue Handle besteht aus einem Ticket dem Thread, der gerade auf dem Handle arbeitet und einem
normalen Handle. Nur wenn True in dem Handle steht, darf jemand auf ihn zugreifen. Steht dort False muss die
entsprechende Funktion failen. -}
type NHandle = (MVar Bool, MVar ThreadId, Handle)
type NSocket = Socket

{- zu anderem Socket verbinden -}
nConnect :: PortHost -> IO (NHandle)
nConnect host =
  do
    handle <- connectTo host epoPortNumber
    ticket <- newMVar True
    thread <- newEmptyMVar
    return (ticket,thread,handle)
```

```
{- Socket belegen, um auf ihm Verbindungen anzunehmen -}
nListen :: IO (NSocket)
nListen =
  do
    socket <- listenOn epoPortNumber
    return socket

{- Verbindung aus Socket annehmen -}
nAccept :: NSocket-> IO (NHandle)
nAccept socket =
  do
    handle <- accept socket
    ticket <- newMVar True
    thread <- newEmptyMVar
    return (ticket,thread,(fst handle))

{- Schreiben auf NHandle -}
nWrite :: NHandle -> String -> IO ()
nWrite (ticket,threadmvar,handle) val =
  do
    waitOn (hPutStrLn handle val) (ticket,threadmvar,handle)

{- Lesen von dem Handle -}
nRead :: NHandle -> IO (String)
nRead (ticket,threadmvar,handle) =
  do
    val <- waitOn (hGetLine handle) (ticket,threadmvar,handle)
    return val

{- Schließen des Handles -}
nClose :: NHandle -> IO ()
nClose (ticket,threadmvar,handle) =
  do
    waitOn (hClose handle) (ticket,threadmvar,handle)
    -- putStrLn "nClose: take"
    dummy <- takeMVar ticket
    -- weitere Operationen auf Handle nicht erlaubt
    -- putStrLn "nClose: putFalse"
    putMVar ticket False

{- Diese Funktion startet die übergebene Routine in einem Thread und wartet, bis dieser fertig ist -}
waitOn :: IO t -> NHandle -> IO t
waitOn thread (ticket,threadmvar,handle) =
  do
    test <- hIsOpen handle
    if test
      then -- nur dann etwas tun
        do
          returnval <- newEmptyMVar
          qsem <- newQSem 0
          forkIO ((doThread thread returnval(ticket,threadmvar,handle))
                          `finally` signalQSem qsem)
          -- putStrLn "Reading QSem"
          waitQSem qsem
          -- putStrLn "Got QSem"
          isempty <- isEmptyMVar threadmvar
```

```
          if (not isempty)
            then -- Dann wurde der Prozess unvorhergesehen beendet
              do
                -- sollte dann leer sein
                -- putStrLn ("waitOn: put False");
                putMVar ticket False
                fail ("waitOn: problems")
            else -- Alles ok
              do
                val <- takeMVar returnval
                return val
      else -- Handle ist zu
        do
          fail ("waitOn: Handle ist closed allready")
  where
    doThread :: IO t -> MVar t -> NHandle -> IO ()
    doThread thread returnmvar(ticket,threadmvar,handle) =
      do
        -- putStrLn "Wait for ticket."
        permission <- takeMVar ticket
        -- putStrLn "Got ticket."
        if permission
          then -- ok
            do
              mythreadid<- myThreadId
              putMVar threadmvar mythreadid
              -- putStrLn "Calling thread..."
              retval <- thread
              -- putStrLn "Thread Called"
              -- Ticket zurückgeben
              putMVar ticket True
              -- Wert zurückgeben
              putMVar returnmvar retval
              dummy <- takeMVar threadmvar
              forget dummy
          else -- keine Erlaubnis
            do
              fail "No Permission to access handle".
```

Bibliothekslisting 6: NSocket.hs

Bibliotheksbaustein 7: ExtPostOffice.hs

```
{- Das externe Postamt
Autor: Ulrich Norbisrath
Erstellungsdatum: 27.04.2000
-}

import IO hiding (try)
import IOExts
import Concurrent
import NSocket
import FiniteMap
import Posix
import List
import Monad
import Exception hiding(try)

import PortGlobals
import ExtPostOfficeConstants

-- tryAllIO ist mächtiger und fängt fail ab
try = tryAllIO
```

```
{- Hier werden die Handles zur Bedienung der einzelnen Verbindungen gespeichert -}
type Port2HandleMVar = MVar (FiniteMap PortDescriptor NHandle)

{- Hier wird eine Liste von Portdescriptoren und Namen gespeichert -}
type Name2PortMVar = MVar (FiniteMap String PortDescriptor)

{- Die inverse Liste -}
type Port2NameMVar = MVar (FiniteMap PortDescriptor String)

{- Eine Liste von Prozessen und den zu Ihnen gehörenden Ports und des letzten heartbeats des Prozesses -}
type Process2Ports = MVar (FiniteMap ProcessID
                              ((MVar [PortDescriptor]),MVar Int))

data GlobalContext =
  GlobalContext { port2Handle :: Port2HandleMVar,
                  name2Port :: Name2PortMVar,
                  port2Name :: Port2NameMVar,
                  process2Ports :: Process2Ports }

{- Verbindung auswerten -}
doCommunicate :: GlobalContext -> NHandle -> IO()
doCommunicate gc handle =
  do
    -- Daten empfangen
    -- putStrLn ("doCommunicate: start")
    lineexc <- try (nRead handle)
    case lineexc of
      (Left _) -> -- exception
        do
          nop
      (Right line) ->
        do
          putStrLn("doCommunicate: " ++ line)
          -- und auswerten
          case (read line) of
            -- Verbindung für ReadPort etablieren
            (EPOConnect pd) ->
              do
                map <- takeMVar (port2Handle gc)
                {- Nachsehen, ob Port vorhanden -}
                case (lookupFM map pd) of
                  Nothing -> -- nicht vorhanden: ok -> Bestät. und Handle abspeichern
                    do
                      putStrLn( "doCommunicate: EPOConnect:" ++
                                "Sending accept" )
                      -- neuen PortHandle in Datenbank aufnehmen
                      putMVar (port2Handle gc) (addToFM map pd handle)
                      -- Teste, ob Prozess in Datenbank, ggf. neu anlegen
                      processmap <- takeMVar (process2Ports gc)
                      case (lookupFM processmap (pProcessId pd)) of
                        Nothing -> -- nein, also neu aufnehmen
                          do
                            cputime <- getCPUTime
                            portlistmvar <- newMVar [pd]
                            cputimemvar <- newMVar cputime
                            putMVar (process2Ports gc)
                                    (addToFM processmap (pProcessId pd)
                                      (portlistmvar, cputimemvar))
```

```
           (Just (portlistmvar,_)) -> -- Ja, dann Port hinzuf.
             do
                portlist <- takeMVar portlistmvar
                putMVar portlistmvar (pd:portlist)
                putMVar (process2Ports gc) processmap
          -- Bestät., Verb. offen lassen
          forgettry (nWrite handle (show EPOAccept))
         -- Handle offen lassen
      (Just _) -> -- Vorhanden, also Problem
        do
          putStrLn
           ("doCommunicate: EPOConnect: Sending reject")
         -- alte Datenbank zurück
          putMVar (port2Handle gc) map
          forgettry (nWrite handle (show EPOReject))
          forgettry (nClose handle) -- Verbindung schließen
(EPOClose pd) ->
  do
    -- versuche Port abzumelden
    test <- destroyPort pd gc
    if test
      then
        do
          putStrLn
           ("doCommunicate: EPOClose: Sending accept")
          forgettry (nWrite handle (show EPOAccept))
      else
        do
          putStrLn
           ("doCommunicate: EPOClose: Sending reject")
          forgettry (nWrite handle (show EPOReject))
    forgettry (nClose handle) -- Verbindung schließen
-- Port unter einem Namen registrieren
(EPORegister pd name) ->
  do
    map <- takeMVar (name2Port gc)
    {- Nachsehen, ob Name bereits vergeben -}
    case (lookupFM map name) of
       Nothing -> -- nicht vorhanden, dann ggf. neuen Eintrag anlegen
        do
         -- Nachsehen, ob Port bereits registriert
          revmap <- takeMVar (port2Name gc)
         case (lookupFM revmap pd) of
            Nothing -> -- nicht vorhanden, ok, dann anlegen
             do
                putMVar (name2Port gc) (addToFM map name pd)
                putMVar (port2Name gc)
                                  (addToFM revmap pd name)
                forgettry (nWrite handle (show EPOAccept))
           (Just _) -> -- schon vorhanden, ablehnen
             do
                putMVar (name2Port gc) map
                putMVar (port2Name gc) revmap
                forgettry (nWrite handle (show EPOReject))
      (Just _) -> -- schon vorhanden, ablehnen
        do
          putMVar (name2Port gc) map
          forgettry (nWrite handle (show EPOReject))
    forgettry (nClose handle) -- Verbindung schließen
-- Nach einem Port zu einem bestimmten Namen suchen
(EPOLookup name) ->
  do
    map <- readMVar (name2Port gc)
```

```
        {- Nachsehen, ob Port zu Namen vorhanden -}
        case (lookupFM map name) of
           Nothing-> -- nicht vorhanden
             do
                forgettry(nWrite handle (show EPOReject))
          (Just pd) -> -- vorhanden, Port schicken
             do
                forgettry(nWrite handle (show (EPOPort pd)))
         forgettry(nClose handle) -- Verbindung schließen
-- Daten an einen Port weiterleiten
(EPOSend pd confirm str) ->
  do
     extSend pd confirm str(port2Handle gc)
     forgettry(nClose handle)
(EPOSendTo name confirm str) ->
  do
     map<- readMVar (name2Port gc)
    {- Nachsehen, ob Port zu Namen vorhanden -}
    case (lookupFM map name) of
       Nothing-> -- nicht vorhanden
         do
           if confirm
             -- Ablehnung
             then forgettry (nWrite handle (show EPOReject))
             else nop -- sonst nichts zurücksenden
      (Just pd) -> -- vorhanden
         do
            extSend pd confirm str(port2Handle gc)
     forgettry(nClose handle)
-- Zeit für einen Prozess neu festsetzen
(EPOLiving pid) ->
  do
     cputime<- getCPUTime
     processmap<- takeMVar(process2Ports gc)
    -- Prozess suchen
    case (lookupFM processmap pid) of
       Nothing-> -- nein, darf eigentlich nur am Anfang vorkommen
         do -- Aber wenn, einfach leeren Eintrag für Prozess kreieren
            portlistmvar<- newMVar []
            cputimemvar<- newMVar cputime
            putMVar(process2Ports gc)
                     (addToFM processmap pid
                        (portlistmvar,cputimemvar))
      (Just (_,cputimemvar)) -> -- Ja, dann cputime erneuern
         do
            dummy<- takeMVar cputimemvar
            forget dummy
            putMVar cputimemvar cputime
            putMVar(process2Ports gc) processmap
-- Port überprüfen
(EPOTest pd) ->
  do
     map<- readMVar (port2Handle gc)
    {- Nachsehen, ob Port vorhanden -}
    case (lookupFM map pd) of
       Nothing-> -- nicht vorhanden: schlecht, also Misserfolg senden
         do
            putStrLn("doCommunicate: EPOTest: port not found")
            forgettry(nWrite handle (show EPOReject))
      (Just chhandle) -> -- Vorhanden, also Erfolg senden
         do
            putStrLn("doCommunicate: EPOTest: port found")
            forgettry(nWrite handle (show EPOAccept))
     forgettry(nClose handle)
```

```
                -- alle anderen Nachrichten ablehnen
                _ ->
                  do
                     forgettry(nWrite handle (show EPOReject))
                     forgettry(nClose handle)  -- Kommunikation beendet -> Handle
                                                -- schließen
  where
    -- Diese Funktion schliesst nicht den Handle
    extSend :: PortDescriptor -> Bool -> String
                              -> Port2HandleMVar -> IO ()
    extSend pd confirm str mapmvar=
      do
        map <- readMVar mapmvar
        {- Nachsehen, ob Port vorhanden -}
        case (lookupFM map pd) of
          Nothing -> -- nicht vorhanden: schlecht, also ggf. ablehnen
            do
              if confirm
                then
                  do
                     putStrLn("doCommunicate: port not found")
                     forgettry(nWrite handle (show EPOReject))
                else nop -- sonst nichts zurücksenden
          (Just chhandle) -> -- Vorhanden, also Daten an chhandle weiterreichen
            do
              putStrLn( "doCommunicate: EPOSend: Sending to"
                         ++ (show pd) ++ ": " ++ (show (EPOWrite str)) )
              test<- try (nWrite chhandle (show (EPOWrite str)))
              putStrLn( "try: done" )
              case test of
                (Right _) -> -- alles ok
                  do
                     putStrLn("doCommunicate: Sending was ok".)
                    if confirm  -- Bestätigung
                      then forgettry (nWrite handle (show EPOAccept))
                      else nop -- sonst nichts zurücksenden
                (Left _) -> -- Probleme, ggf. exception auswerten???
                  do
                     putStrLn("doCommunicate: Sending was not ok".)
                    -- Da es Probleme gab, muss der Port abgemeldet werden
                     destroyPort pd gc
                    if confirm -- Ablehnung senden
                      then forgettry (nWrite handle (show EPOReject))
                      else nop -- sonst nichts zurücksenden

{- Port aus Datenbanken austragen. Liefert True bei erfolgreicher Abmeldung. -}
destroyPort :: PortDescriptor -> GlobalContext -> IO Bool
destroyPort pd gc=
  do
    putStrLn ("destroying Port: " ++ (show pd))
    map <- takeMVar (port2Handle gc)
    {- Nachsehen, ob Port vorhanden -}
    case (lookupFM map pd) of
      Nothing -> -- nicht vorhanden, ignorieren, da also schon weg
        do
          -- alte Datenbank zurück
           putMVar(port2Handle gc) map
          return False
      (Just chhandle) -> -- Vorhanden
        do
          -- Teste, ob der Port registriert wurde
           revmap<- takeMVar (port2Name gc)
           map' <- takeMVar (name2Port gc)
```

```
        case (lookupFM revmap pd) of
          Nothing -> -- nicht registriert, also nichts entfernen
            do
              putMVar(port2Name gc) revmap -- alte DB zurück
              putMVar(name2Port gc) map' -- alte DB zurück
          (Just name) -> -- fündig -> diesen Namen löschen
            do
              putMVar(port2Name gc) (delFromFM revmap pd)
              putMVar(name2Port gc) (delFromFM map' name)
        -- Eintrag aus Prozessliste entfernen
        processmap <- readMVar (process2Ports gc)
        case (lookupFM processmap (pProcessId pd)) of
          Nothing -> -- nein, das kann nicht sein
            do -- wenn, dann ignorieren
              nop
          (Just (portlistmvar,_)) -> -- Ja, dann Port löschen
            do
              portlist <- takeMVar portlistmvar
              putMVar portlistmvar (delete pd portlist)
        -- Eintrag in Handle-Datenbank löschen
        putMVar (port2Handle gc) (delFromFM map pd)
        -- Terminiermeldung auf Handle senden
        forgettry (nWrite chhandle (show EPOTerminate))
        -- Handle des Ports schließen
        putStrLn "destroyPort: Closing port-handle"
        forgettry (nClose chhandle)
        putStrLn "destroyPort: Handle closed"
        return True

{- Der Prozess, der überwacht, dass die Client-Prozesse noch leben -}
lifeChecker :: GlobalContext -> IO ()
lifeChecker gc =
  do
    threadDelay epoLifeCheckTime
    -- Teste Zeiten der Prozesse
    processmap <- readMVar (process2Ports gc)
    -- Bestimme die Prozesse, deren Zeiten abegelaufen sind
    cputime <- getCPUTime -- aktuelle Zeit
    -- gehe alle Prozesse durch und baue dabei eine neue Liste auf
    newlist <- (filterM (testProcess cputime) (fmToList processmap))
    -- putMVar (process2Ports gc) (listToFM newlist)
    lifeChecker gc -- und weiter
  where
    -- Folgende Funktion schließt alle Ports eines Prozesses, wenn dessen Zeit
    -- abgelaufen ist
    testProcess :: Int
                -> (ProcessID, (MVar [PortDescriptor], MVar Int))
                -> IO Bool
    testProcess cputime (processid, portlisttime) =
      do
        portlist <- readMVar (fst portlisttime)
        oldtime <- readMVar (snd portlisttime)
        --putStrLn ("testProcess: newtime: "++(show cputime)
        --        ++" oldtime: "++(show oldtime)
        --        ++" portlist: "++(show portlist))
        -- wegen Verschiebungen doppelte Zeit zulassen
        if (cputime - oldtime) > (div epoLifeCheckTime 500000)
          then
            do
              -- destroyAll portlist
              foldM (\x y -> dodestroy y) () portlist
              return False
          else
              return True
```

```
    dodestroy :: PortDescriptor -> IO ()
    dodestroy pd =
      do
        dummy <- destroyPort pd gc
        forget dummy
    destroyAll :: [PortDescriptor] -> IO ()
    destroyAll ([]) =
      do
        return ()
    destroyAll (head:tail) =
      do
        dummy <- destroyPort head gc
        forget dummy
        print tail
        destroyAll tail

{- Verbindungen annehmen -}
takeConnections :: GlobalContext -> NSocket -> IO()
takeConnections gc socket =
  do
    -- putStrLn ("takeConnections: Waiting for connection...")
    exchandle <- try (nAccept socket)
    -- putStrLn ("takeConnections: Accepted connection.")
    case exchandle of
      (Left _) -> -- exception Aufgetreten
        do
          putStrLn ("takeConnections: Exception occured"
                    ++ " while listening on socket".)
      (Right handle) -> -- Verbindung korrekt angenommen
        do
          -- putStrLn ("takeConnections: forking thread with Handle.")
          forkIO ( doCommunicate gc handle) -- Verbindung weiterverarb.
          -- putStrLn ("takeConnections: forked thread with Handle.")
          nop
    takeConnections gc socket

main =
  do
    -- Zugriffe auf offene Pipes ignorieren
    installHandler sigPIPE Ignore Nothing
    -- Socket öffnen
    socket <- nListen
    portmap <- newMVar emptyFM -- Liste zur Speicherung der Channels der Ports
    registermap <- newMVar emptyFM -- Liste zur Speicherung der Namen der Ports
    reverseregistermap <- newMVar emptyFM -- Umgekehrte Liste nach Ports sort.
    portslistmvar <- newMVar emptyFM
    let gc = (GlobalContext portmap registermap
                            reverseregistermap portslistmvar)
    -- Lifechecker starten
    forkIO (lifeChecker gc)
    -- Verbindungen annehmen
    forgettry (takeConnections gc socket)

{- Da getCPUTime immer 10000000000 liefert, muss ich es selbst programmieren -}
getCPUTime :: IO Int
getCPUTime =
  do
    time <- epochTime
    return time
```

```
{- Versuch ohne Auswertung -}
forgettry exp =
  do
    test <- (try exp)
    case test of
     (Right _) ->
       nop -- alles ok
     (Left _) ->
       putStrLn "forgettry: warning: IO-Exception occured".
```

Bibliothekslisting 7: ExtPostOffice.hs

Bibliotheksbaustein 8: Makefile

```
# Haskell-Compiler
HC      = ghc

# utils
MAKEDEP = mkdependHS
RM = rm
AR = ar
CP = cp

# Besondere Optionen
EXTRA_HC_OPTS =

# Haskell Optionen
HC_OPTS = -syslib concurrent\
         -syslib net\
         -syslib data\
         -syslib util\
         -syslib posix\
         -cpp \
         $(EXTRA_HC_OPTS)

# Library
LIBSRCS = PortGlobals.hs \
         ExtPostOfficeConstants.hs\
         NSocket.hs\
         ExtPostOfficeAccess.hs\
         IntPostOffice.hs\
         Port.hs
LIBHI = $(LIBSRCS:.hs=.hi)
LIBOBJS = $(LIBSRCS:.hs=.o)
LIBPATH = ./lib
LIBNAME = port
LIB = $(LIBPATH)/lib$(LIBNAME).a

# Interface, was man von außen noch braucht
INTERFACE = $(LIBPATH)/PortGlobals.hi $(LIBPATH)/Port.hi

# externes Postamt
EPOSRCS = ExtPostOffice.hs
EPOOBJS = $(EPOSRCS:.hs=.o)
EPOPATH = ./bin
EPO = $(EPOPATH)/ExtPostOffice

# alle Sourcen
SRCS = $(LIBSRCS) $(EPOSRCS)
OBJS = $(SRCS:.hs=.o)

LIBS = -L$(LIBPATH) -l$(LIBNAME)

.SUFFIXES : .o .hi .lhs .hc .s
```

```
all: lib $(EPO)

lib: $(LIB)
$(LIB): $(LIBOBJS) $(INTERFACE)
        $(AR) r $(LIB) $(LIBOBJS)

$(LIBPATH)/%.hi: $(LIBOBJS) ;$(CP) $*.hi $(LIBPATH)

dep: .depend
depend: .depend
.depend: $(SRCS)
        $(MAKEDEP) -- $(HC_OPTS) -- \
              -x Exception -x IOExts -x Weak -f .depend $(SRCS)

$(EPO) : $(LIBOBJS) $(EPOOBJS) $(LIB) $(INTERFACE)
        $(HC) -o $@ $(HC_OPTS) $(LIBS) $(EPOOBJS)

# o-Files bekommt man aus hs-Files durch Compilieren
%.o: %.hs; $(HC) -c $< $(HC_OPTS)

# hi-Files bekommt man aus o-Files
%.hi: %.o %.hs

clean:
        $(RM) -f a.out *.hi *.o .depend* $(LIB) $(EPO) $(INTERFACE)

include .depend
```

Bibliothekslisting 8: Makefile für Bibliothek

Quelltextverzeichnis

Listing 1.1: Kleines I/O-Beispiel 5
Listing 1.2: Kleines I/O-Beispiel in do-Notation 5
Listing 1.3: Kleines I/O-Beispiel mit return 5
Listing 1.4: Exceptions 6
Listing 1.5: Kleines I/O-Beispiel mit Überwachung des Einlesens von der Standardeingabe 6
Listing 5.1: Einfaches nebenläufiges System 30
Listing 5.2: Zitat aus Chatclient aus Kapitel 8.2 31
Listing 6.1: Beispiel für interne Zeichenkettenübergabe 40
Listing 6.2: Beispiel für Wichtigkeit der Kontrolle der Thread-ID zur Laufzeit 41
Listing 6.3: Internal Port 42
Listing 6.4: PortDescriptor 42
Listing 6.5: PortInfo 42
Listing 6.6: PortChannel 43
Listing 6.7: MergePort 43
Listing 6.8: Komplette Datenstruktur Port 44
Listing 7.1: testinternal.hs 68
Listing 7.2: testextern.hs 69
Listing 8.1: Interface.hs (Schnittstelle des Datenbankbeispiels) 72
Listing 8.2: Server.hs (Server des Datenbankbeispiels) 73
Listing 8.3: Client.hs (Client des Datenbankbeispiels) 75
Listing 8.4: Interface.hs (Schnittstelle des Chatbeispiels) 76
Listing 8.5: Server.hs (Beispiel eines Chatservers) 77
Listing 8.6: Client.hs (Beispiel eines Chatclients) 79
Bibliothekslisting 1: PortGlobals.hs 86
Bibliothekslisting 2: IntPostOffice.hs 89
Bibliothekslisting 3: ExtPostOfficeConstants.hs 91
Bibliothekslisting 4: ExtPostOfficeAccess.hs 95
Bibliothekslisting 5: Port.hs 108
Bibliothekslisting 6: NSocket.hs 110
Bibliothekslisting 7: ExtPostOffice.hs 117
Bibliothekslisting 8: Makefile für Bibliothek 118

Abbildungsverzeichnis

Abbildung 1.1: Rechner, Prozesse, Threads und Kommunikation 8
Abbildung 3.1: Channel 1 20
Abbildung 3.2: Channel 2 21
Abbildung 3.3: Channel 3 21
Abbildung 3.4: Channel 4 21
Abbildung 3.5: unReadChan 1 22
Abbildung 3.6: unReadChan 2 22
Abbildung 3.7: unReadChan 3 22
Abbildung 4.1: MWMR Consumer-Producer. Anfangssystem 24
Abbildung 4.2: MWMR Consumer-Producer. System nach Ausfall des Puffers 25
Abbildung 4.3: MWMR Consumer-Producer. Rückführung in konsistenten Zustand 26
Abbildung 4.4: MWMR Consumer-Producer. Konistenter Zustand mit neuem Puffer 26
Abbildung 4.5: MWSR Consumer-Producer. Anfangssystem 27
Abbildung 4.6: MWSR Consumer-Producer. System nach Ausfall des Puffers 27
Abbildung 4.7: MWSR Consumer-Producer. Rückführung in konsistenten Zustand 27
Abbildung 4.8: MWSR Consumer-Producer. Konsistenter Zustand mit neuem Puffer 28
Abbildung 6.1: Aufbau eines Readports 41
Abbildung 6.2: Vereinfachte Darstellung eines Readports 41
Abbildung 6.3: Rechner 1 startet Prozess 1 44
Abbildung 6.4: Thread 1 kreiert Readport 1 45
Abbildung 6.5: Start Thread 2, Übergabe Readport, Umwandlung in Writeport 45
Abbildung 6.6: Registrieren des Readport 1 unter »Name« 45
Abbildung 6.7: Registrieren des Readport 1 unter »Name« 46
Abbildung 6.8: Prozess 2 auf Rechner 1 wird gestartet 46
Abbildung 6.9: Thread 1 in Prozess 2 legt benannten Writeport »Name« an 47
Abbildung 6.10: Thread 1 in Prozess 2 schreibt Daten auf lokalen benannten Writeport »Name« 47
Abbildung 6.11: Rechner 2 startet Prozess 1 48
Abbildung 6.12: Thread 1 kreiert benannten Writeport mit Ziel (Rechner 1, Name) 48
Abbildung 6.13: Thread 1 von Prozess 1 auf Rechner 1 schreibt Daten an Port mit Name »Name« auf Rechner 1 49

Quellenverzeichnis

[AAr98] T. Arts, J. Armstrong. *A practical type system for Erlang*. Technical report, Erlang User Conference, Sept. 1998

[ASG] Abraham Silberschatz, Peter Bear Galvin. *Operating System Concepts*. Fith Edition.Addison-Wesley Publishing Company, 1998

[AVW96] Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams. *Concurrent Programming in ERLANG*. Second Edition. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[BLO96] S. Breitinger, R. Loggen, Y. Ortega-Mallén, R. Peña-Mari. *Eden -- The paradise of Functional Concurrent Programming. Europar 1996, Springer LNCS 1123*

[CGK98] Manuel M. T. Chakravarty, Yike Guo, Martin Köhler. *Distributed Haskell: Goffin on the Internet*. Proceedings of the Third Fuji International Symposium on Functional and Logic Programming, World Scientific, 1998

[GHC] The Glasgow Haskell Compiler. http://www.haskell.org/ghc

[Han97] Michael Hanus. *A Unified Computation Model for Declarative Programming.* In 1997 Joint Conference on Declarative Programming (APPIA-GULP-PRODE'97), 1997

[Han99] Michael Hanus. *Ditributed Programming in a Multi-Paradigm Declarative Language*. Springer LNCS 1702, pp. 188-205, 1999

[Has] The Haskell Homepage. http://www.haskell.org

[HLi] The GHC Team. *Haskell Libraries 4.06.*

[HPF99] Paul Hudak, John Peterson, Joseph Fasel. *A Gentle Introduction to Haskell 98*. `http://www.haskell.org/tutorial`, September 28, 1999

[Huc99] Frank Huch. *Erlang-Style Distributed Haskell.* Implementation of Functional Languages, 11th. International Workshop (IFL'99). September 7th - 10th, Lochem. Draft Proceedings

[HUG] Hugs - *A Haskell Interpreter.*`http://www.haskell.org/hugs`

[JGF96] SL Peyton Jones, A Gordon, S Finne. *Concurrent Haskell.* 23rd ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida, Jan 1996, pp295-308.

[JHA99] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, Philip Wadler. *The Haskell 98 Report*. Feb 1999. `http://www.haskell.org`

[JHB99] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, Philip Wadler. *The Haskell Library Report 1.4*. Feb 1999. `http://www.haskell.org`

[JME] Simon Peyton Jones, Simon Marlow, and Conal Elliott. *Stretching the storage manager: weak pointers and stable names in Haskell.*

[KDE] *KDE*. `http://www.kde.org`

[LHE93] Lektorat des BI-Wiss. Verl. unter Leitung von Hermann Engesser. *Duden Informatik*. Dudenverlag. Zweite Auflage. Mannheim 1993.

[MWa97] S. Marlow, P. Wadler. *A practical subtyping system for Erlang*. In Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming, pages 136-149, Amsterdam, 9-11 June 1997

[Ned] Nedit. `ftp://ftp.fnal.gov/pub/nedit`

[RHi96] Ralf Hinze. *Online Haskell Kurs.* `http://www.informatik.uni-bonn.de/~ralf/Hskurs_toc.html`

[Ski99] Jan Skibinski. *Haskell Companion*. Okt 1999.

[SOf] *StarOffice*. `http://www.sun.com/staroffice`

[SuS] *SuSE Linux*. `http://www.suse.de`

[VMw] *VMware*. `http://www.vmware.com`

Index

<!> 34
<!!> 35
<|> 33
»start« 71
A
action 4f
Anforderungsdefinition 29
Anleitung 67
Athlon 83
Aufbau der Arbeit 2
Ausgabedaten 2
Auswertungsstrategie 4
B
Bankserver 1
Bestätigungsmeldung 35
Bindeglied 38
Bool 3
Boolean 3
C
call-by-need 4
call-by-value 9
Channel 20, 24
Channels 8
Chatserver und -client 75
completePort 60
concurrent constraint 11
Concurrent Haskell 17
conditional rule 13
Curry 13
D
data 3
Datenbank 71
Datenbankserver 71
Datenkapselung 32
Datentyp 3
Deadlock 14, 64
Deadlocks 10
Debugger 64
destroyPort 60
Distributed Haskell 11
do 5
E
Eden 12
Eingabedaten 2
Einleitung 1
Either 3
epo 50
EPO 50
EPOAccept 51
EPOClose 51
epoConnect 55
EPOConnect 51
epoHeartBeat 55, 57
epoLifeCheckTime 55
EPOLiving 52
epoLookup 56
EPOLookup 52
EPOMessage 51
EPOPort 52
epoRegister 56
EPORegister 52
EPOReject 52
epoRemove 56
epoSend 56
EPOSend 52
epoSendTo 56
EPOSendTo 52
epoTest 55
EPOTest 52
EPOTransferMessage 57
Eq 59
equational constraints 11, 13
Ericsson 9
Erlang 9
Erlang-Style Distributed Haskell 14
ErrorCall 6
Erweiterbarkeit 32
Exception 5, 36
External Post Office 50
externes Postamt 38
Externes Postamt 50
ExtPostOffice.hs 54, 110
ExtPostOfficeAccess.hs 55, 91
ExtPostOfficeConstants 51
ExtPostOfficeConstants.hs 51, 90
EXTRA_HC_OPTS 70
EXTRA_LIBS 70
F
fac 4
fail 6
Fakultätsfunktion 4
Fensteroberfläche: 83
forkIO 17, 36
funktionale Programme 2

Funktionale Programme 3
funktionale Programmiersprache 2
Funktionenräumen 2

G

getCpuTime 64
getHostEntry 64
getLine 4
ghc 5, 83
GlobalContext 54
Goffin 11
Großbuchstabe 3

H

Haskell 2
Heavy-Weight-Processes 7

I

imperative Programmiersprachen 5
Implementation 37
indirekte interne Kommunikation 46
InternalPort 39, 59
Internal Ports 39
Internal Post Office 57
internes Postamt 38
Internes Postamt 57
IntPostOffice.hs 57, 86
IO-Monade 4
IPO 57
ipoChannel 60
IPOChannelType 57
ipoConnect 58
ipoInternalThread 57f
ipoLookup 58
IPOLookup 58
IPOMap 57
IPOMessage 57
ipoRemove 58
IPORemove 58
IPOStore 58
IPOTerminate 57
IPOTransferMessage 57
IPOWrite 57
isEmptyMVar 19

J

Just 3

K

KDE 83
Kernel: 83
killThread 17, 36
Kommunikationsverbindungen 7
Konstruktoren 3

L

Lambda-Abstraktion 5
Left 3
Light-Weight-Processes 7
link 35
Link 32
Link-Mechanismus 10
Linux 83
lookupLocalPort 34
lookupPort 32, 34, 38f, 61

M

Makefile 117
Marshalling 38
Maybe 3
mergePort 32f, 38f, 59
MergePort 39, 59
MergePortMessage 59
MergePortMessages 59
Mergeports 43
mkdependHS 67
Multiple-Writer-Multiple-Reader-Konzept 23
Multiple-Writer-Single-Reader-Konzept 23
MWMR 23
MWSR 23, 26
myIPAddress 50
myThreadId 18, 36

N

nAccept 53
Name2PortMVar 54
nClose 54
nConnect 53
nebenläufig 7
newChan 21
newEmptyMVar 19
newMVar 19
newPort 32f, 38, 60
newQSem 20
newRefNr 59
NHandle 53
nListen 53
Nomenklatur 7
Nothing 3
nRead 54
NSocket.hs 53, 108
nWrite 54

O

OBJS 68, 70
Ord 59

P

Pattern Matching 4
pid 10
Port 28, 31, 38, 42, 59
Port.hs 59, 95
Port2HandleMVar 54
Port2NameMVar 54
Portbasierte Prozesse 7
PortChannel 43, 59
port constraints 12f
Portdescriptor 41f, 49
PortDescriptor 42
PortGlobals.hs 49, 85

porthost 32
PortHost 32
PortID 41
Port-ID 41
PortInfo 42, 59
portname 32
PortName 32
PORTPATH 68, 70
Portschnittstelle 59
Port t 31
Process2Ports 54
PROGS 68, 70
Prozess 7
Prozess-ID 7
putMVar 19
putStrLn 4

Q

Quelltexte 85
Quelltexteditor 83
Queues 8

R

raiseInThread 18, 36
Read 31f, 61
readChan 21
Reader 7, 24
readMVar 19
Readonlyports 33
readPort 34
Readport 33, 39f
readPortTimed 34
Rechner 7
refNrCounter 59
refNRCounter 60
register 10
registerPort 32ff, 61
Rekursion 2
return 5
Right 3

S

Schnittstelle 38
sendToLocalPort 35
sendToLocalPortFast 35
sendToPort 35
sendToPortFast 35
Show 31f
Signalhandler 54
signalQSem 20
Simulationsumgebung 83
Socketzugriffsfunktionen 53
spawn 10
SRCS 70
StarOffice 83
start 71, 75
Subprozessen 2
Suse 83
swapMVar 19

T

takeMVar 19
temporal constraints 11, 13f
threadDelay 18, 36
Threadid 36
ThreadKilled 18
Threads 2, 7
Timestamp 55
trace 64
try 5f, 36
tryAllIO 36
Typchannel 40
type 3f

U

unendlich 4
unGetChan 21
ungetypt 9
unlink 35
unsafePerformIO 60
unSafePerformIO 59

V

VMware 7, 83

W

waitOn 53
waitQSem 20
Wordprocessor 83
World 4
writeChan 21
writePort 34
writePortFast 35
Writeports 33
Writer 7, 24

Y

yield 18, 36

Z

Zeichen 7
Zugriffsfunktionen 34
Zustandswandler 4