Fortgeschrittene Techniken der Funktionalen Programmierung

Abstrakte Systemprogrammierung

Christoph Lüth

http://www.informatik.uni-bremen.de/~cxl/

WS 04/05

Einführung und Rückblick

Organisatorisches

- Vorlesung: Mi 17-19 MZH 8090.
- Übung: selbstbestimmt
- Scheinkriterien: drei Übungsblätter, eine Semesteraufgabe.
- Entschuldigung im voraus.

Themen

- Monaden: was ist eigentlich eine Monade, die `IO` und `ST` Monaden, Reader/Writer-Monaden und andere;
- nebenläufige Programmierung (Concurrent Haskell)
- Ausnahmen, asymetrische Ausnahmen, und Ausnahmebehandlung;
- language interoperability und das Foreign Function Interface (FFI): von Haskell weg (call-out), nach Haskell hin (call-in);
- Template Haskell;
- Feinheiten des Typsystems:
  - existentielle Typen, höhere Polymorphie und andere sexy types;
- Vielleicht. . . GUIs mit HTk oder wxHaskell, oder grafische Programmierung mit hOpenGL;

Ressourcen

- Haskell-Webseite: `http://www.haskell.org/`
- Büchereien:
  - Haskell 98 Libraries (standardisiert)
  - Haskell Hierarchical Libraries (nicht standardisiert, aber Standard)
- Compiler:
  - Glasgow Haskell Compiler (`ghc`) (Version 6.2)

Rückblick

Die Funktionale Programmiersprache Haskell:

- Definition von Funktionen:
  - lokale Definitionen mit `let` und `where`
  - Fallunterscheidung und guarded equations
  - Abseitsregel
  - Funktionen höherer Ordnung
- Typen:
  - Basisdatentypen: `Int`, `Integer`, `Rational`, `Double`, `Char`, `Bool`
  - Strukturierte Datentypen: `[a]`, `(a, b)`
  - Algebraische Datentypem: `data Maybe a = Just a | Nothing`

Rückblick (2)

Die Funktionale Programmiersprache Haskell:

- Abstrakte Datentypen
- Module
- Typklassen
- Verzögerte Auswertung und unendliche Datentypen

Der IO Monade

I/O in funktionalen Sprachen

- Problem:

  Eingabe kann nicht als Funktion `readLine ::() -> String` modelliert werden — zerstört referentielle Transparenz.

- Generelles Problem hier: Interaktion mit der Umwelt
- Mögliche Lösungen:
  - Seiteneffekte (e.g. Standard ML);
  - Continuations (Auswertungskontext explizit modellieren);
  - Streams: `readLine :: Instream-> (Instream, String)`
  - Einkapselung in Monaden (Haskell).

Monadische I/O

- Abstrakter Datentyp `IO a` :

```
(>>=)  :: IO t-> (t-> IO u)-> IO u  -- "then"
return :: t-> IO t                  -- "return"
```

- `t :: IO a` $\equiv$ erst eine Aktion, gibt dann Wert vom Typ `a` zurück:

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

Monadische I/O

- Elementare Operationen:

```
getLine  :: IO String        -- eine Zeile lesen
putStr   :: String-> IO ()   -- Zeile ausgeben
putStrLn :: String-> IO ()   -- Zeile mit LF ausgeben
```

- “Einmal I/O, immer I/O”
- Abhängigkeit von Umwelt am Typ erkennbar
- Daher:

```
main :: IO ()
```

  Hauptprogramm hat keinen Rückgabewert, nur noch Interaktion.

Monadische I/O: Die `do` Notation

- Syntaktischer Zucker für Monaden:

```
echo = do s<- getLine          echo = getLine
          putStrLn s    <-->           >>= \s -> putStrLn s
          echo                         >> echo
```

- Oder auch:

```
echo = do { s<- getLine; putStrLn s; echo }
```

- Nützlich:

```
(>>) :: IO t-> IO u-> IO u
f >> g == f >>= \_ -> g
```

Monadische I/O: Einfache Beispiele

```
echo :: IO ()
echo = getLine >>= putStrLn >> echo
echo = do { l <- getLine; putStrLn l; echo }

interactOnce :: (String-> String)-> IO ()
interactOnce f = getLine >>= (putStr . f)
interactOnce f = do { l<- getLine; putStrLn (f l) }

revecho :: IO ()
revecho = getLine >>= putStrLn . reverse >> revecho
revecho = do { l<- getLine; putStrLn (reverse l); revecho }
```

File I/O

Abstrakter Zugriff durch lazy evaluation:

```
type FilePath  = String
getContents    :: IO String
readFile       :: FilePath -> IO String
writeFile      :: FilePath -> String -> IO ()
appendFile     :: FilePath -> String -> IO ()
```

Beispiel:

```
cntWords :: FilePath -> IO ()
cntWords file = do c <- readFile file
                   let s = (length . words) c
                   putStr $ file++ " : "++ show s++ " words\n"
```

Fortgeschrittene File I/O

```
data IOMode    = ReadMode | WriteMode | AppendMode
openFile       :: FilePath -> IOMode -> IO Handle

hGetContents   :: Handle -> IO String  -- uvm.
hFlush         :: Handle -> IO ()

hGetPosn       :: Handle -> IO HandlePosn
hSetPosn       :: HandlePosn -> IO ()

data SeekMode = AbsoluteSeek | RelativeSeek | SeekFromEnd
hSeek          :: Handle -> SeekMode -> Integer -> IO ()
```

Weitere übliche Operationen (Buffering etc) siehe Haskell98 Library Report, Kap. 11.

Fehler!

Repräsentation durch den abstrakten Datentyp `IOError`.
Ausnahmebehandlung ähnlich in Java:

```
ioError  :: IOError -> IO a
catch    :: IO a -> (IOError -> IO a)-> IO a
```

Beispiel:

```
cntW file = catch (cntWords file)
                  (\e-> putStr ("Error: " ++ (show e)))
```

Analyse der Fehler durch `isDoesNotExistsError :: IOError-> Bool`
etc.

Kommandozeilenargumente

Interaktion mit der Umgebung: Modul `System`

```
data ExitCode = ExitSuccess | ExitFailure Int
getArgs       :: IO [String]
getProgName   :: IO String
getEnv        :: String -> IO String
system        :: String -> IO ExitCode
exitWith      :: ExitCode -> IO a
```

Beispiel:

```
main = do r<- getProgName; a<- getArgs
          catch (mapM_ cntWords a)
                (\e-> putStrLn (r++ ": "++ (show e)))
```

Das Modul Directory

```
createDirectory                 :: FilePath -> IO ()
removeDirectory, removeFile     :: FilePath -> IO ()
renameDirectory, renameFile     :: FilePath -> FilePath -> IO ()

getDirectoryContents :: FilePath -> IO [FilePath]
getCurrentDirectory  :: IO FilePath
setCurrentDirectory  :: FilePath -> IO ()

data Permissions = ...
readable, writeable, executable, searchable :: Permissions-> F
getPermissions       :: FilePath -> IO Permissions
setPermissions       :: FilePath -> Permissions -> IO ()
getModificationTime  :: FilePath -> IO ClockTime
```

Das Modul Directory, Beispiel

```
import Directory
import Time
import System(getArgs)

cleanup dir =
 do now<- getClockTime
    c<- getDirectoryContents dir
    setCurrentDirectory dir
    mapM_ (\f-> do {mt<- getModificationTime f;
                    if ((last f == '~') &&
                        tdDay (diffClockTimes mt now) >= 1)
                    then removeFile f else return ()}) c
main = do { d<- getArgs; cleanup (head d) }
```

Systemfunktionen für Haskell

- Abstrakte Modellierung in Haskell98 Standard Library:
  `IO`, `Directory` `System`, `Time`
  Siehe Library Report
- Konkrete Modellierung in Modul Posix (nur für `GHC`) nach IEEE Standard 1003.1, e.g.:

```
executeFile :: FilePath                          -- Command
            -> Bool                              -- Search PATH?
            -> [String]                          -- Arguments
            -> Maybe [(String, String)]          -- Environment
            -> IO ()
```

More IO

Nützliche Kombinatoren (aus dem `Prelude`):

```
sequence    :: [IO a] -> IO [a]
sequence_   :: [IO a] -> IO ()
mapM        :: (a -> IO b) -> [a] -> IO [b]
mapM_       :: (a -> IO b) -> [a] -> IO ()
```

Mehr im Modul `Monad` (Library Report, Kapt. 10).

Zusammenfassung

- Abhängigkeit von Aussenwelt in Typ IO kenntlich
- Benutzung von IO: vordefinierte Funktionen in der Haskell98 Bücherei
- Nächstes Mal:
  - Was steckt dahinter?
  - Flucht aus Alcatraz – IO für Erwachsene
  - Endlich Variablen

Exklusiv:
Monads — The Inside Story

Tagesprogramm

Heute in diesem Theater:

- Die Geheimnisse der Monaden
- Endlich Zuweisung
- Flucht aus Alcatraz – `IO` für Erwachsene

Zustandsübergangsmonaden

- Aktionen (`IO a`) sind keine schwarze Magie.
- Grundprinzip: Systemzustand Σ wird explizit behandelt.

`f :: a-> IO b` $\cong$ `f :: (a, Σ) -> (b, Σ)`

Zustandsübergangsmonaden

- Aktionen (`IO a`) sind keine schwarze Magie.
- Grundprinzip: Systemzustand Σ wird explizit behandelt.

  `f :: a-> IO b` $\cong$ `f :: (a, Σ) -> (b, Σ)`

  Folgende Invarianten müssen gelten:

  - Systemzustand darf nie dupliziert oder vergessen werden.
  - Auswertungsreihenfolge muß erhalten bleiben.

- Komposition muss Invarianten erhalten.

  $\rightsquigarrow$ Zustandsübergangsmonaden

Zustandsübergangsmonaden

- Typ:

```
type ST s a = s-> (a, s)
```

$$\texttt{a -> ST s b} \cong \texttt{a-> s-> (b, s)} \cong \texttt{(a, s)-> (b, s)}$$

Parametrisiert über Zustand `s` und Berechnungswert `a`.

- Komposition durch

```
(>>=) :: ST s a-> (a-> ST s b)-> ST s b
```

Komposition von Zustandsübergängen

- Im Prinzip Vorwärtskomposition `(>.>)`:

```
(>>=) :: ST s a-> (a-> ST s b)-> ST s b
(>>=) :: (s-> (a, s))-> (a-> s->  (b, s))-> (s-> (b, s))
(>>=) :: (s-> (a, s))-> ((a, s)-> (b, s))-> (s-> (b, s))
```

- Damit `f >>= g = uncurry g . f`.
- Aber: `ST` kann kein Typsynonym sein

  abstrakter Datentyp um Invarianten zu erhalten)

ST als Abstrakter Datentyp

- Datentyp verkapseln:

```
newtype ST s a = ST (s-> (a, s))
```

- Hilfsfunktion (Selektor)

```
unwrap :: ST s a-> (s-> (a, s))
unwrap (ST f) = f
```

- Damit ergibt sich

```
instance Monad (ST s) where
  f >>= g = ST (uncurry (unwrap. g) . unwrap f)
  return  a = ST (\s-> (a, s))
```

Aktionen

- Aktionen: Zustandstransformationen auf der Welt
- Typ `RealWorld#` repräsentiert Außenwelt
  - Typ hat genau einen Wert `realworld#`, der nur für initialen Aufruf erzeugt wird.
  - Aktionen: `type IO a = ST RealWorld# a`
- Optimierungen:
  - `ST s a` durch in-place-update implementieren.
  - `IO`-Aktionen durch einfachen Aufruf ersetzen.
    - ▷ Compiler darf keine Redexe duplizieren!
  - Typ `IO` stellt lediglich Reihenfolge sicher.

Was ist eigentlich ein Monad?

`ST` modelliert imperative Konzepte.

Beobachtung: Andere Konzepte können ähnlich modelliert werden:

- Nichtdeterminismus: $f : a \Rightarrow \mathbb{P}(b)$ mit Komposition

  $f : \mathbb{P}(a) \Rightarrow (a \Rightarrow \mathbb{P}(b)) \Rightarrow (\mathbb{P}(b))$

  $A\texttt{>>=}f = \{fa \mid a \in A\}$

- Ausnahmen: `f :: a-> Maybe b` mit Komposition

```
(>>=) :: Maybe a-> (a-> Maybe b)-> Maybe b
Just a  >>= f = f a
Nothing >>= f = Nothing
```

Was ist eigentlich ein Monad?

`ST` modelliert imperative Konzepte.

Beobachtung: Andere Konzepte können ähnlich modelliert werden:

- Nichtdeterminismus: $f : a \Rightarrow \mathbb{P}(b)$ mit Komposition

  $f : \mathbb{P}(a) \Rightarrow (a \Rightarrow \mathbb{P}(b)) \Rightarrow (\mathbb{P}(b))$

  $A \texttt{>>=} f = \{f\,a \mid a \in A\}$

- Ausnahmen: `f :: a-> Maybe b` mit Komposition

```
(>>=) :: Maybe a-> (a-> Maybe b)-> Maybe b
Just a  >>= f = f a
Nothing >>= f = Nothing
```

- Benötigen Typklassen für Typkonstruktoren. . .

Konstruktorklassen

- Monaden sind Konstruktorklassen:
  Typklassen für Typkonstruktoren (kinds)
- Beispiel:

```
class Functor f where
  fmap :: (a-> b) -> (f a -> f b)

instance Functor [] where
  fmap f [] = []
  fmap f (x:xs) = f x : map f xs
```

Monads: The Inside Story

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
    (>>)   :: m a -> m b -> m b
    fail   :: String -> m a

    p >> q  = p >>= \_ -> q
    fail s  = error s
```

Folgende Gleichungen müssen (sollten) gelten:

```
return a >>= k  ≡  k a
  m >>= return  ≡  m
m >>= ( x-> k x >>= h)  ≡  (m >>= k)>>= h
```

Beispiel: Referenzen

Speicher und Referenzen.

Signatur:

```
type Mem
instance Mem Monad

newRef   :: Mem Ref
readRef  :: Ref -> Mem String
writeRef :: Ref -> String-> Mem ()
```

Implementation der Referenzen

Speicher: Liste von Strings, Referenzen: Index in Liste. Zeigen.

```
type Mem = ST [String]   -- Zustand
type Ref = Int

newRef = ST (\s-> (length s, s++[""]))
readRef r = ST (\s-> (s !! r, s))
writeRef r v = ST (\s-> ((),
                  take r s ++ [v]++ drop (r+1) s))

run :: Mem a-> a
run (ST f)= fst (f [])
```

IORef — Referenzen

Datentyp der Standardbücherei (GHC, Hugs)

```
import Data.IORef

data IORef a

newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()

modifyIORef :: IORef a -> (a -> a) -> IO ()
atomicModifyIORef :: IORef a -> (a -> (a, b)) -> IO b
```

Implementation: “echte” Referenzen.

Beispiel: Referenzen

```
fac :: Int-> IO Int
fac x = do acc <- newIORef 1
           loop acc x where
             loop acc 0 =  readIORef acc
             loop acc n = do t<- readIORef acc
                             writeIORef acc (t* n)
                             loop acc (n-1)
```

Zeigen.

Ausbruch aus Alcatraz

- Aus dem `IO`-Monaden gibt es keinen Ausweg.
  - Im Gegensatz zu z.B. `Maybe`:

```
fromMaybe :: a -> Maybe a -> a
```

- Das ist manchmal unpraktisch: Initialisierungen etc.
- Für `ST` gibt es

```
fixST :: (a -> ST s a) -> ST s a    -- Fixpunkt
runST :: (forall s . ST s a) -> a   -- NB: Typ!
```

- Für `IO` gibt es . . .

Unsichere Aktionen

```
import System.IO.Unsafe(unsafePerformIO)

unsafePerformIO :: IO a-> a
```

Warnung: gefährlich und nicht typsicher!

```
test :: IORef [a]
test = unsafePerformIO $ newIORef []

main = do writeIORef test [42]
          bang <- readIORef test
          putStrLn (bang :: [Char])
```

Verwendung von `unsafePerformIO`

Nur verwenden bei:

- Einmalig durchgeführter, von anderen IO-Aktionen unabhängiger Aktion (zum Beispiel Konfigurationsdatei lesen).
- `IO`-Aktionen, die nur einmal durchgeführt werden sollen.
- Alloziierung globaler Ressourcen (z.B. Referenzen).
- Debugging (traces, logfiles).

Verwendung von `unsafePerformIO`

Alloziierung globaler Referenzen:

```
-- | Generate a new identifier.
newId :: IO Int
newId = do atomicModifyIORef r (\i-> (i+1, i))
           where r = unsafePerformIO (newIORef 1)
{-# NOINLINE newId #-}
```

NOINLINE beachten — Optimierungen verhindern.

Debugging:

```
trace :: String -> a-> a
trace s x = unsafePerformIO (putStrLn s >>= return x)
```

aber auch vordefiniert (`Debug.Trace`).

Zusammenfassung & Ausblick

- Blick hinter die Kulissen von `IO`
- Monaden und andere Kuriositäten
- Referenzen
- `unsafePerformIO`
- Nächstes Mal: Nebenläufigkeit und Monadentransformer ?!?

Mehr über Monaden

Heute gibt's:

- Monaden, mehr Monaden, und noch mehr Monaden,
- Kombination von Monaden,
- Fallbeispiel: monadische Parserkombinatoren.

Der Identitätsmonade

Der allereinfachste Monade:

```
type Id a = a

instance Monad Id where
  return a = a
  b >>= f  = f b
```

Der Fehlermonade

`Maybe` kennt nur `Nothing`, daher:

```
data Either a b = Left a  | Right b

instance Monad (Either String) where
   (Right a) >>= f     = f a
   (Left l)  >>= f     = Left l
   return b            = Right b
```

Der Listenmonade

Listen sind eine Instanz von `Monad`

```
instance Monad [] where
    m >>= f  = concatMap f m
    return x = [x]
    fail s   = []
```

Intuition: `f :: a-> [b]` Liste der möglichen Resultate

Der Lesemonade

```
newtype Reader e a = Reader (e -> a)

instance Monad (Reader e) where
    return a          = Reader $ \e -> a
    (Reader r) >>= f = Reader $ \e -> let Reader g = f (r e)
                                      in  g e
```

- Intuition: Werte der Eingabe `e` lesen und verabeiten.
- Eingabe wird nicht modifiziert.
  - Nicht der Fall bei Lesen aus Datei
- Beispiel: Lesen aus einer Symboltabelle

Der Schreibemonade

- Produziert einen Strom von Werten.

```
newtype Writer w a = Writer (a,[w])

instance Monad (Writer w) where
  return a               = Writer (a, [])
  (Writer (a,w)) >>= f = let Writer (a',w') = f a
                         in Writer (a', w++ w')
```

- Kein Zugriff auf geschriebene Werte möglich
- Beispiel: “Logging”

Der Continuationmonade

- Auswertungskontext wird explizit modelliert.

```
newtype Cont r a = Cont { runCont :: ((a -> r) -> r) }
```

- `r` ist der Typ der gesamten Berechnung
- `a-> r` ist der momentane Kontext

```
instance Monad (Cont r) where
    return a        = Cont $ \k -> k a
    (Cont c) >>= f = Cont $ \k -> c (\a -> runCont (f a) k)
```

- `GOTO` für funktionale Sprachen.
- Lieber nicht benutzen.

Kombination von Monaden: Das Problem

Gegeben zwei Monaden:

```
class Monad m1 where ...
```

```
class Monad m2 where ...
```

Dann gelten weder

```
class Monad (m1 (m2 a))
class Monad (m2 (m1 a))
```

Problem: Monadengesetze gelten nicht.

Eine Lösung: Monadentransformer

- Monadentransformer: Monade mit "Loch"
- Beispiel: Zustandsmonadentransformer

```
type StateT s m a =  s-> m (s, a)
```

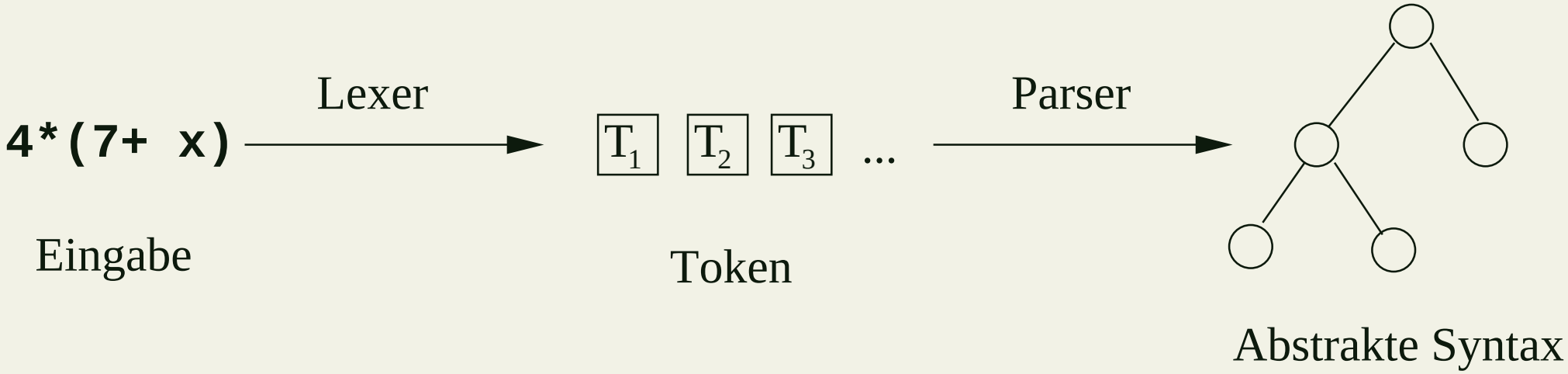
  - Zustandsbasierte Berechnungen in einem anderen Monaden `m`
  - `StateT s Id` ist Zustandsmonade
- Nicht sehr praktisch: alle Kombinationen müssen definiert werden.
  - Kombination von `State` und `Error`:

```
type ErrorState s a = Either String (s-> (s ,a))
type StateError s a = s-> (s, Either String a)
```

- Praktischer Ansatz: alles in den `IO`-Monaden.

Monadische Parserkombinatoren

- Gegeben: Grammatik
- Gesucht: Funktion, die Wörter der Grammatik erkennt



- Parser bildet Eingabe auf Parsierungen ab.
- Basisparser erkennen Terminalsymbole
- Kombinatoren:
  - Sequenzierung (erst A, dann B)
  - Alternierung (entweder A oder B)
  - Abgeleitete Kombinatoren (z.B. Listen A^*, nicht-leere Listen A^+)

  $\Longrightarrow$ Nichtterminalsymbole

Arithmetische Ausdrücke

- Grammatik:

```
Expr   ::= Term + Term
         | Term - Term
         | Term
Term   ::= Factor * Factor
         | Factor / Factor
         | Factor
Factor ::= Number | (Expr)
Number ::= Digit  | Digit Number
Digit  ::= 0 |...| 9
```

- Daraus abstrakte Syntax:

```
data Expr   = Plus  Expr Expr
            | Minus Expr Expr
            | Times Expr Expr
            | Div   Expr Expr
            | Number Int
            deriving (Eq, Show)
```

  - Hier Unterscheidung `Term`, `Factor`, `Number` unnötig.

Modellierung in Haskell

- Welcher Typ für Parser?
  - Parser übersetzt Token in abstrakte Syntax
  - Parametrisiert über Eingabetyp (Token) `a` und Ergebnis `b`

Modellierung in Haskell

- Welcher Typ für Parser?
  - Parser übersetzt Token in abstrakte Syntax
  - Parametrisiert über Eingabetyp (Token) `a` und Ergebnis `b`
  - Müssen mehrdeutige Ergebnisse modellieren

Modellierung in Haskell

- Welcher Typ für Parser?
  - Parser übersetzt Token in abstrakte Syntax
  - Parametrisiert über Eingabetyp (Token) `a` und Ergebnis `b`
  - Müssen mehrdeutige Ergebnisse modellieren
  - Müssen Rest der Eingabe modellieren

Modellierung in Haskell

- Welcher Typ für Parser?
  - Parser übersetzt Token in abstrakte Syntax
  - Parametrisiert über Eingabetyp (Token) `a` und Ergebnis `b`
  - Müssen mehrdeutige Ergebnisse modellieren
  - Müssen Rest der Eingabe modellieren

```
type Parse a b = [a]-> [(b, [a])]
```

- Beispiel: `parse "3+4*5"` $\rightsquigarrow$
```
[ (3, "+4*5"),
  Plus 3 4, "*5"),
  (Plus 3 (Times 4 5), "")]
```

Basisparser

- Erkennt nichts:

```
none :: Parse a b
none = const []
```

- Erkennt alles:

```
suceed :: b-> Parse a b
suceed b inp = [(b, inp)]
```

- Erkennt einzelne Zeichen:

```
token :: Eq a=> a-> Parse a a
token t = spot (== t)
spot  :: (a-> Bool)-> Parse a a
spot p []     = []
spot p (x:xs) = if p x then [(x, xs)] else []
```

Kombinatoren

- Alternierung:

```
infixl 3 `alt`
alt :: Parse a b-> Parse a b-> Parse a b
alt p1 p2 i = p1 i ++ p2 i
```

- Sequenzierung:
  - Rest des ersten Parsers als Eingabe für den zweiten

```
infixl 5 >*>
(>*>) :: Parse a b-> Parse a c-> Parse a (b, c)
(>*>) p1 p2 i = [((y, z), r2) | (y, r1) <- p1 i,
                                (z, r2) <- p2 r1]
```

- Eingabe weiterverarbeiten:

```
infix 4 `build`
build :: Parse a b-> (b-> c)-> Parse a c
build p f inp = [(f x, r) | (x, r)<- p inp]
```

- Damit z.B. Sequenzierung rechts/links:

```
infixl 5 *>, >*
(*>) :: Parse a b-> Parse a c-> Parse a c
(>*) :: Parse a b-> Parse a c-> Parse a b
p1 *> p2 = p1 >*> p2 `build` snd
p1 >* p2 = p1 >*> p2 `build` fst
```

Abgeleitete Kombinatoren

- Listen: $A^* ::= AA^* \mid \varepsilon$

```
list :: Parse a b-> Parse a [b]
list p = p >*> list p `build` uncurry (:)
         `alt` suceed []
```

- Nicht-leere Listen: $A^+ ::= AA^*$

```
some :: Parse a b-> Parse a [b]
some p = p >*> list p `build` uncurry (:)
```

- NB. Präzedenzen: `>*>` (5) vor `build` (4) vor `alt` (3)

Der Kern des Parsers

- Parsierung von `Expr`

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pTerm
                              `build` uncurry Plus
        `alt` pTerm >* token '-' >*> pTerm
                              `build` uncurry Minus
        `alt` pTerm
```

- Parsierung von `Term`

```
pTerm :: Parse Char Expr
pTerm = pFactor >* token '*' >*> pFactor
                              `build` uncurry Times
        `alt` pFactor >* token '/' >*> pFactor
                                 `build` uncurry Div
        `alt` pFactor
```

- Parsierung von `Factor`

```
pFactor :: Parse Char Expr
pFactor =
       some (spot isDigit) `build` Number. read
       `alt` token '(' *> pExpr >* token ')'
```

Die Hauptfunktion

- Lexing: Leerzeichen aus der Eingabe entfernen
- Zu prüfen: Testen.
  - Parsierung braucht Eingabe auf
  - Keine Mehrdeutigkeit

```
parse :: String-> Expr
parse i =
  case filter (null. snd)
         (pExpr (filter (not.isSpace) i)) of
    [] -> error "Input does not parse."
    [(e, _)] -> e
    _  -> error "Ambiguous input."
```

Ein kleiner Fehler

- Mangel: 3+4+5 ist Syntaxfehler
- Behebung: leichte Änderung der Grammatik . . .

```
Expr   ::= Term + Expr | Term - Expr | Term
Term   ::= Factor * Term | Factor / Term | Factor
Factor ::= Number | (Expr)
Number ::= Digit  | Digit Number
Digit  ::= 0 |...| 9
```

  - (vergleiche alt)
- Abstrakte Syntax bleibt . . .

- Entsprechende Änderung des Parsers in `pExpr`

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pExpr
                            `build` uncurry Plus
        `alt` pTerm >* token '-' >*> pExpr
                            `build` uncurry Minus
        `alt` pTerm
```

  - (vergleiche alt)

- . . . und in `pTerm`:

```
pTerm :: Parse Char Expr
pTerm = pFactor >* token '*' >*> pTerm
                          `build` uncurry Times
        `alt` pFactor >* token '/' >*> pTerm
                            `build` uncurry Div
        `alt` pFactor
```

- `pFactor` und Hauptfunktion bleiben: Testen.

Monadische Parserkombinatoren

- Der Parser ist ein Monade:

```
instance Monad (Parser a) where
  return a = \inp-> [(a, inp)]
  f >>= g  = \inp-> concat [g v out | (v, out)<- f inp]
```

- Explizite Parametrisierung über dieser Monade erlaubt Zustand beim Parsieren:
  - Symboltabellen
  - Zeilennummern

Zusammenfassung Parserkombinatoren

- Systematische Konstruktion des Parsers aus der Grammatik.
- Durch verzögerte Auswertung annehmbare Effizienz.
- Einfache Implementierung (wie oben) skaliert nicht
- Grammatik muß eindeutig sein (LL(1) o.ä.)
- Gut implementierte Büchereien (wie Parsec) bei eindeutiger Grammatik auch für große Eingaben geeignet

Zusammenfassung

- Monaden sind praktische Abstraktion
- Kombination bereitet Probleme, daher `IO` (die Mutter aller Monaden)
- Parserkombinatoren sind das Mittel der Wahl für flexible, performante Parser
  - `Parsec` mit dem GHC
- Ausblick: Nebenläufigkeit — Concurrent Haskell.

Nebenläufige Programmierung in Haskell: Grundlagen

Heute gibt's hier: Nebenläufigkeit

- Grundkonzepte
- Implementation in Haskell
- Basiskonzepte

Konzepte der Nebenläufigkeit

Thread (lightweight process)	vs.	Prozess
Programmiersprache		Betriebssystem
(z.B. Java, Haskell)		
gemeinsamer Speicher		getrennter Speicher
mehrere pro Programm		einer pro Programm

Multitasking:

- präemptiv: Kontextwechsel wird erzwungen
- kooperativ: Kontextwechsel nur freiwillig

Zur Erinnerung: Threads in Java

- Erweiterung der Klassen `Thread` oder `Runnable`
- Gestartet wird Methode `run()` — durch eigene überladen
- Starten des Threads durch Aufruf der Methode `start()`
- Kontextwechsel mit `yield()`
- Je nach JVM kooperativ oder präemptiv.
- Synchronisation mit `synchronize`
  - Fehlerhaft implementiert/spezifiziert?

Threads in Haskell: Concurrent Haskell

- Sequentielles Haskell: Reduktion eines Ausdrucks
- Nebenläufiges Haskell: Reduktion eines Ausdrucks an mehreren Stellen
- `ghc` (und `hugs`) implementieren Haskell-Threads
- `ghc`: präemptiv, `hugs`: kooperativ
- Modul `Concurrent` enthält Basisfunktionen
- Wenige Basisprimitive, darauf aufbauend Abstraktionen

Concurrent Haskell

- Jeder Thread hat einen Identifier — `type ThreadId`
- Neuen Thread erzeugen: `forkIO:: IO()-> IO ThreadId`
- Thread stoppen: `killThread :: ThreadId -> IO ()`
- Kontextwechsel: `yield :: IO ()`
- Eigener Thread: `myThreadId :: IO ThreadId`
- Warten: `threadDelay :: Int -> IO ()`
  - Argument in Mikrosekunden
  - Auflösung $\cong 50ms$
- Blockierung:
  - Blockierende Systemaufrufe blockieren alle Threads
  - Aber: Haskell Standard-IO blockiert nur den aufrufenden Thread

Concurrent Haskell — erste Schritte

Ein einfaches Beispiel:

```
import Concurrent

write :: Char-> IO ()
write c = putChar c >> write c

main :: IO ()
main = forkIO (write 'X') >> write 'O'
```

Übersetzen: `ghc -package concurrent simple1.lhs`

Mit Hugs keine besonderen Optionen nötig.

Zeigen.

Synchronisation mit MVars

- Basisynchronisationmechanismus in `Concurrent Haskell`
  - Alles andere abgeleitet.
- `MVar a` veränderbare Variable (ähnlich `IORef a`)
- Entweder leer oder gefüllt mit einem `a`
- Verhalten beim Lesen und Schreiben

Zustand vorher:	leer	gefüllt
Lesen	blockiert (bis gefüllt)	danach leer
Schreiben	danach gefüllt	blockiert (bis leer)

- Neue Variable erzeugen (leer oder gefüllt):

```
newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
```

- Lesen:

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

- Schreiben:

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

- Nicht-blockierendes Lesen/Schreiben:

```
tryTakeMVar :: MVar a -> IO (Maybe a)
tryPutMVar  :: MVar a-> a-> IO Bool
```

- Test (Achtung: Zustand kann sich ändern)

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

Ein einfaches Beispiel ohne Synchronisation

- Nebenläufige Eingabe von der Tastatur

```
echo :: String-> IO ()
echo p =
  do putStrLn ("\nPlease enter line for "++p)
     line <- getLine
     randomRIO (1,100) >>= \n-> forN n (putStr (p++ ":"++ line++" "))
     echo p
main :: IO ()
main = forkIO (echo "1") >> echo "2"
```

- Problem: gleichzeitige Eingabe Zeigen

Ein einfaches Beispiel ohne Synchronisation

- Nebenläufige Eingabe von der Tastatur

```
echo :: String-> IO ()
echo p =
  do putStrLn ("\nPlease enter line for "++p)
     line <- getLine
     randomRIO (1,100) >>= \n-> forN n (putStr (p++ ":"++ line++" "))
     echo p
main :: IO ()
main = forkIO (echo "1") >> echo "2"
```

- Problem: gleichzeitige Eingabe Zeigen
- Lösung: `MVar` synchronisiert Eingabe

Ein einfaches Beispiel mit Synchronisation

- `MVar` voll $\Leftrightarrow$ Eingabe möglich
  - Also: initial voll
- Inhalt der MVar irrelevant: `MVar ()`

```
echo :: MVar ()-> String-> IO ()
echo flag p =
  do takeMVar flag
     putStrLn ("\nPlease enter line "++p)
     line <- getLine
     putMVar flag ()
     randomRIO (1,100) >>= \n-> forN n (putStr (p++ ":"++ li
     -- yield  -- noetig, da sonst keine Fairness.
     echo flag p
```

Das Standardbeispiel: Die speisenden Philosophen

- Philosoph i:
  - vor dem Essen i-tes und $(i+1) \bmod n$-tes Stäbchen nehmen
  - nach dem Essen wieder zurücklegen
- Stäbchen modelliert als `MVar ()`

- i-ter Philosoph:

```
philo :: [MVar ()] -> Int-> IO ()
philo chopsticks i =
   let num_phil = length (chopsticks)
   in do putStrLn ("Phil #"++(show i)++" thinks...")
         randomRIO (500,2000) >>= threadDelay
         takeMVar (chopsticks !! i)
         takeMVar (chopsticks !! ((i+1) `mod` num_phil))
         putStrLn ("Phil #"++(show i)++" eats...")
         randomRIO (500,2000) >>= threadDelay
         putMVar (chopsticks !! i) ()
         putMVar (chopsticks !! ((i+1) `mod` num_phil)) ()
         philo chopsticks i
```

- Hauptfunktion: n Stäbchen erzeugen
- Anzahl Philosophen in der Kommandozeile

```
main :: IO ()
main = do num <- getArgs >>= \a-> return (read (head a))
          chopsticks <- sequence
                          (take num (repeat (newMVar ())))
          mapM_ (forkIO . (philo chopsticks)) [0.. num-1]
          block
```

- Hilfsfunktion `block`: blockiert aufrufenden Thread

```
block :: IO ()
block = newEmptyMVar >>= takeMVar
```

- NB: Hauptthread terminiert — Programm terminiert! Zeigen.

Zusammenfassung

Concurrent Haskell bietet

- Threads auf Quellsprachenebene
- Synchronisierung mit `MVars`
- Durch schlankes Design einfache Implementierung
- Funktionales Paradigma erlaubt Abstraktionen (nächste Woche!)

Nebenläufige Programmierung in Haskell: Abstraktionen

Und das erleben Sie heute:

- Letztes Mal: Grundlagen der nebenläufigen Programmierung
  - Threads in Haskell
  - `MVar`
- Heute: darauf aufbauende Abstraktionen
- Längeres Beispiel: Netzwerkprogrammierung

Abstraktionen

Mit `MVars` mächtigere Synchronisationskonzepte implementieren:

- Puffernde Kanäle
- Semaphoren (allgemein, quantitativ)
- Monitore und Mutexe

Buffer, Kanäle, Semaphoren schon im Modul `Concurrent`

Abstraktion I: Kanäle

- Ein Kanal besteht aus Strom mit einem Lese- und Schreibende:

```
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
```

  - Hier `MVar`, damit Lesen/Schreiben nicht unterbrochen wird
- Ein Strom ist `MVar (ChItem a)`:
  - entweder leer,
  - oder enthält Werte aus Kopf `a` und Rest.

```
type Stream a = MVar (ChItem a)
data ChItem a = ChItem a (Stream a)
```

- Schnittstelle:

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

- Neuen Kanal erzeugen:

```
newChan :: IO (Chan a)
newChan =
  do hole  <- newEmptyMVar
     read  <- newMVar hole
     write <- newMVar hole
     return (Chan read write)
```

  - NB: Leseende = Schreibende

In einen Kanal schreiben

- Neues Ende (`hole`) anlegen
- Wert in altes Ende schreiben
- Zeiger auf neues Ende setzen

```
writeChan (Chan _ write) val =
  do new_hole <- newEmptyMVar
     old_hole <- takeMVar write
     putMVar write new_hole
     putMVar old_hole (ChItem val new_hole)
```

- Kann nicht blockieren — `write` immer gefüllt.

Aus Kanal lesen

- Anfang auslesen, Anfangszeiger weitersetzen
- Kann blockieren (wenn Kanal leer)

```
readChan (Chan read _) =
  do read_end <- takeMVar read
     (ChItem val new_read_end)
              <- readMVar read_end
     putMVar read new_read_end
     return val
```

  - `readMVar :: MVar a-> IO a` liest `MVar`, schreibt Wert zurück.
  - `readMVar` statt `takeMVar`, um Duplikation zu ermöglichen

Weitere Kanalfunktionen

- Zeichen wieder vorne einhängen:

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

- Kanal duplizieren (Multicast):

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

- Kanalinhalt als (unendliche) Liste:

```
getChanContents :: Chan a -> IO [a]
```

  - Auswertung terminiert nicht, sondern blockiert

Abstraktionen II: Semaphoren

- Abstrakter Datentyp `QSem`
- Betreten kritischer Abschnitt (P): `waitQSem :: QSem -> IO ()`
- Verlassen kritischer Abschnitt (V): `signalQSem :: QSem -> IO ()`
- Semaphore: Zähler plus evtl. wartende Threads
  - P erniedrigt Zähler, blockiert ggf. aufrufenden Thread
  - V erhöht Zähler, gibt ggf. blockierte Threads frei
- Implementierung von Semaphoren mit `MVar`: eigenes Scheduling
- Variation: Quantitative Semaphoren
  - Zähler kann um Parameter n erhöht/erniedrigt werden

Semaphoren: die P-Operation

```
data QSem = QSem (MVar (Int, [MVar ()]))
```

- `MVar ..` für die ganze Semaphore, darin:
- Zähler der Prozesse im kritischen Abschnitt
- Liste von wartenden Prozessen (`MVar ()`)

```
newQSem :: Int -> IO QSem
newQSem n = do m <- newMVar (n, [])
               return (QSem m)
```

Semaphoren: die P-Operation

Eintritt in kritischen Abschnitt:

- Wenn Eintritt möglich, Zähler erniedrigen
- Ansonsten blockieren (Reihenfolge!)

```
waitQSem :: QSem -> IO ()
waitQSem (QSem sem) = do
   (avail,blocked) <- takeMVar sem
   if avail > 0 then
     putMVar sem (avail-1,[])
    else do
     block <- newEmptyMVar
     putMVar sem (0, blocked++[block])
     takeMVar block
```

Semaphoren: die V-Operation

- Falls wartende threads, einen aufwecken.
- Alternatives Scheduling:
  - am Anfang hinzufügen, vom Anfang nehmen (einfacher, unfair)
  - am besten: zufällige Auswahl

```
signalQSem :: QSem -> IO ()
signalQSem (QSem sem) = do
   (avail,blocked) <- takeMVar sem
   case blocked of
     [] -> putMVar sem (avail+1,[])
     (block:blocked') -> do
           putMVar sem (0,blocked')
           putMVar block ()
```

Abstraktion III: Monitore und Mutexe

- Monitore schützen Aktionen.
- Modellierung als Typklasse:

```
class Synchronize v where
   synchronize :: v -> IO a -> IO a
```

  - Parameter `v`: Datenstruktur, über der synchronisiert wird
  - Beispiel: Semaphoren

```
instance Synchronize QSem where
   synchronize s a = do waitQSem s
                        r<- a
                        signalQSem s
                        return r
```

Reentrante Mutexe

- Problem:
  - Semaphoren (u.a.) nicht reentrant
  - keine Rekursion möglich
- Lösung: reentrantes Mutex (mutual exclusion)
- Operationen: `acquire` und `release`
- Mutex enthält:
  - `ThreadId` des besitzenden Threads, Zähler für Rekursionstiefe
  - Liste von blockierten Threads (`Id`, `MVar`)

```
data Mutex = Mutex (MVar (Maybe (ThreadId,Int),
                                [(ThreadId, MVar ())]))
```

Eintritt in den kritischen Abschnitt

- Mutex akquirieren:

```
acquire :: Mutex -> IO ()
```

  - Wenn Thread Mutex besitzt, dann Zähler erhöhen;
  - sonst neue `MVar ()` erzeugen, ThreadId und MVar in Liste einhängen, auf MVar warten;

Eintritt in den kritischen Abschnitt

```
acquire :: Mutex -> IO ()
acquire (Mutex mvar) =
   do st <- takeMVar mvar
      current <- myThreadId
      case st of
         (Nothing,[]) -> putMVar mvar (Just (current,1),[])
         (Just (holder,n),pnd) ->
            if current == holder
              then putMVar mvar (Just (holder,n+1),pnd)
              else do bsem <- newEmptyMVar
                      putMVar mvar (Just (holder,n),
                                    (current,bsem):pnd)
                      takeMVar bsem
```

Verlassen des kritischen Abschnitts

- Mutex freigeben:

```
release :: Mutex -> IO ()
```

  - Prüfen, ob Thread Mutex besitzt (sonst Fehler)
  - Wenn Zähler 1 und Liste leer, wird Mutex leer
  - Wenn Zähler 1, dann ersten Thread aus Liste aufwecken
  - Wenn Zähler größer 1, dann erniedrigen

Verlassen des kritischen Abschnitts

```
release :: Mutex -> IO ()
release (Mutex mvar) =
   do st <- takeMVar mvar
      current <- myThreadId
      case st of
         (Just (h,n),pnd) | current == h ->
            release' mvar h n pnd
         _ -> do putMVar mvar st
                 error "Illegal lock release"  where
    release' mvar _ 1 [] = putMVar mvar (Nothing,[])
    release' mvar _ 1 ((h',sem):pnd') =
         do putMVar mvar (Just (h',1),pnd'); putMVar sem ()
    release' mvar h n pnd = putMVar mvar (Just (h,n-1), pnd)
```

Reentrante Monitore

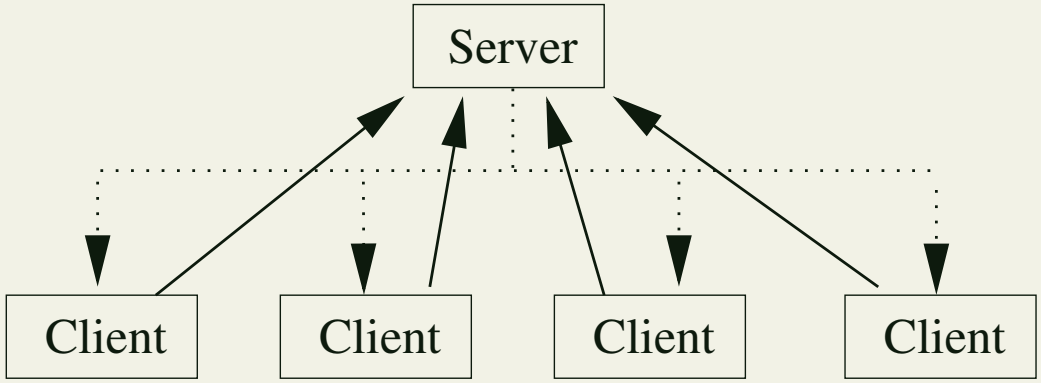
Mit Mutex Implementierung reentranter Monitore:

```
instance Synchronize Mutex where
  synchronize m a =
    do acquire m
       r<- a
       release m
       return r
```

Rekursion möglich (aber teuer).

Längeres Beispiel: It's good to talk

- Ziel: ein Programm, um sich über das Internetz zu unterhalten (`talk`, `IRC`, etc.)
- Verteilte Architektur:



- Hier: Implementierung des Servers
  - Netzverbindungen durch Socket

Socketprogrammierung

- Socket erzeugen, an Namen binden, mit `listen` Verbindungsbereitschaft anzeigen
- Zustandsbasierte Verbindung:
  - Serverseite: mit `accept` auf eingehende Verbindungen warten
  - Jede Verbindung erzeugt neuen Filedescriptor
    $\Rightarrow$ inhärent nebenläufiges Problem!
  - Clientseite: Mit `connect` Verbindung aufnehmen.
- Zustandslose Verbindung: `sendTo` zum Senden, `recvFrom` zum Empfangen.
- GHC-Modul `Network`
  - Low-level Funktionen in `Network.Socket`

Das Modul Network

```
type Socket
data PortID = Service String -- z.B. "ftp"
            | PortNumber PortNumber
            | UnixSocket String -- Socket mit Namen (im Date
type Hostname = String
instance Num PortNumber
-- Zustandsbasiert:
listenOn   :: PortID-> IO Socket
accept     :: Socket-> IO (Handle, Hostname, PortNumber)
connectTo  :: Hostname -> PortID -> IO Handle
-- Zustandslos:
sendTo   :: HostName -> PortID -> String -> IO ()
recvFrom :: HostName -> PortID -> IO String
```

Serverarchitektur

- Ein Kanal zur Nachrichtenverbreitung:
  - eine Nachricht, viele Empfänger (broadcast)
  - Realisierung mittels `dupChan`
- Zentraler Scheduler
- Für jede ankommende Verbindung neuer Thread:
  - Nachrichten vom Socket auf den Kanal schreiben
  - Nachrichten vom Kanal in den Socket schreiben
- Problem: Wie aus Socket oder Kanal lesen wenn beide blockieren?

Serverarchitektur

- Ein Kanal zur Nachrichtenverbreitung:
  - eine Nachricht, viele Empfänger (broadcast)
  - Realisierung mittels `dupChan`
- Zentraler Scheduler
- Für jede ankommende Verbindung neuer Thread:
  - Nachrichten vom Socket auf den Kanal schreiben
  - Nachrichten vom Kanal in den Socket schreiben
- Problem: Wie aus Socket oder Kanal lesen wenn beide blockieren?
- Lösung: Zwei Threads

Serverarchitektur

- Ein Kanal zur Nachrichtenverbreitung:
  - eine Nachricht, viele Empfänger (broadcast)
  - Realisierung mittels `dupChan`
- Zentraler Scheduler
- Für jede ankommende Verbindung neuer Thread:
  - Nachrichten vom Socket auf den Kanal schreiben
  - Nachrichten vom Kanal in den Socket schreiben
- Problem: Wie aus Socket oder Kanal lesen wenn beide blockieren?
- Lösung: Zwei Threads
- Clientarchitektur: `telnet`

Talk 0.1: Hauptprogramm

```
main :: IO ()
main =
  do portNum <- getArgs >>= return . read . head
     s <- listenOn (PortNumber (fromInteger portNum))
     ch<- newChan
     loop s ch
```

Talk 0.1: Hauptschleife

```
loop :: Socket -> Chan String -> IO ()
loop s ch =
  do (handle, wh, p) <- accept s
     hSetBuffering handle NoBuffering
     putStrLn ("New connection from "++ wh++ " on "++ show p)
     ch2    <- dupChan ch
     forkIO (newUser handle ch2)
     loop s ch
```

Talk 0.1: Benutzerprozess

```
newUser :: Handle-> Chan String -> IO ()
newUser socket msgch =
  forkIO read >> write where
    read :: IO ()
    read  = hGetLine socket >>= writeChan msgch >> read
    write :: IO ()
    write = readChan msgch >>= hPutStrLn socket >> write
```

Zeigen!

Talk 0.1: Zusammenfassung

Übersetzen mit `-package concurrent -package net`

Nachteile:

- Nachrichten stauen sich im Kanal
- Serverprozess fällt um, wenn Benutzer Verbindung beendet
- Keine Fehlerbehandlung
- Benutzer anonym

Zusammenfassung

- Abstraktionen über einfachen `MVars`:
  - Kanäle
  - Semaphoren
  - Monitore
- Haskell: the world's finest concurrent programming language?
- Längeres Beispiel: Talk.

Exceptions:
Die Ausnahmen von der Regel

Heute wartet auf Sie:

- Letztes Mal: ein fehleranfälliger `talk`-Server
- Heute: Fehlerbehandlung
- Ausnahmebehandlung in Haskell98
- Unscharfe Ausnahmen
- Dynamische Ausnahmen
- Achtung, Ausnahmen!

Ausnahmebehandlung in Haskell98

Haskell 98: Fehler leben im IO-Monaden.

- Fehler fangen:

```
catch    :: IO a -> (IOError -> IO a)-> IO a
```

  - Variante: `try :: IO a -> IO (Either IOError a)`

- Fehler erzeugen:

```
userError :: String-> IOError
ioError   :: IOError-> IO a
```

  Oder durch andere Operationen im IO-Monaden.

Fehler analysieren

Funktionen, die im Handler benutzt werden können:

```
isAlreadyExistsError  :: IOError -> Bool
isDoesNotExistError   :: IOError -> Bool
isAlreadyInUseError   :: IOError -> Bool
isFullError           :: IOError -> Bool
isEOFError            :: IOError -> Bool
isIllegalOperation    :: IOError -> Bool
isPermissionError     :: IOError -> Bool
isUserError           :: IOError -> Bool

ioeGetErrorString     :: IOError -> String
ioeGetHandle          :: IOError -> Maybe Handle
ioeGetFileName        :: IOError -> Maybe FilePath
```

Talk 0.2: Hauptprogramm

```
main :: IO ()
main =
  do port_num <- getArgs >>= return . read . head :: IO Intege
     s <- listenOn (PortNumber (fromInteger port_num))
     ch <- newChan
     loop s ch
```

- Kanäle nach Bedarf erzeugen

Talk 0.2: Hauptschleife

```
loop :: Socket -> Chan String -> IO ()
loop s ch =
  do (handle, wh, p) <- accept s
     hSetBuffering handle NoBuffering
     installHandler sigPIPE Ignore Nothing
     putStrLn ("New connection from "++ wh++
               " on port "++ show p)
     ch2 <- dupChan ch
     forkIO (catch (newUser handle wh ch2)
                   (\_ -> hClose handle))
     loop s ch2
```

- Fehlerbehandlung für `newUser`, `SIGPIPE` ignorieren

Talk 0.2: Benutzerprozess

```
newUser :: Handle-> String-> Chan String -> IO ()
newUser socket wh msgch =
  do hPutStrLn socket "Hello there. Please send your nickname.
     nick <- do nm <- hGetLine socket
                return (filter (not . isControl) nm)
     hPutStrLn socket ("Nice to meet you!")
     writeChan msgch (nick ++ "@" ++ wh ++ " has joined.")
```

Talk 0.2: Benutzerprozess (Forts.)

```
  wp <- forkIO write
  catch (read ((nick ++ ": ")++)) $
   \e-> do killThread wp
           if isEOFError e then
             writeChan msgch (nick++ "@"++ wh++ " has left.
             else writeChan msgch (nick++ "@"++ wh++
                                   " left hastily ("++
                                   ioeGetErrorString e++ ")
           hClose socket where
read :: (String-> String)-> IO ()
read f = hGetLine socket >>= writeChan msgch. f >> read f
write :: IO ()
write = readChan msgch >>= hPutStrLn socket >> write
```

Talk 0.2: Zusammenfassung

Nachteile:

- Kein Nachrichtenstau mehr
- Fehlerbehandlung für Benutzerprozess
- Anmeldeprozedur: Benutzer hat Namen

Talk 0.2: Zusammenfassung

Nachteile:

- Kein Nachrichtenstau mehr
- Fehlerbehandlung für Benutzerprozess
- Anmeldeprozedur: Benutzer hat Namen
- Schnell verkaufen!

Probleme mit der Ausnahmebehandlung in Haskell98

- Keine Ausnahmebehandlung für rein funktionalen Code.
  - `error :: String-> a` bricht Programmausführung ab;
  - z.B. Fehler bei `read :: Read a=> String-> a`?
    - `readIO :: Read a=> String-> IO a` wirft Ausnahme
  - Laufzeitfehler (pattern match, fehlende Klassenmethoden, . . .)
- Keine Behandlung von asynchronen Ausnahmen möglich.
  - Nebenläufige Fehler, e.g. stack overflow, Speichermangel, Interrupts;

Probleme mit rein funktionalen Ausnahmen.

Warum nicht einfach `throw :: Exception-> a`?

- Wird die Ausnahme geworfen?

  ```
  length [throw exception]
  ```

  Abhängig von Tiefe der Auswertung (wertet `length` Argument aus?)

- Welche Ausnahme wird geworfen:

  ```
  throw ex1 + throw ex2
  ```

  Abhängig von Reihenfolge der Auswertung der Argumente

NB: Haskell98 ist spezifiert als nicht-strikt, aber nicht als *lazy*.

Auswertungsreihenfolge unspezifiziert!

Unscharfe Ausnahmen.

- Normale Ausnahmen:

  Wert eines Ausdrucks = Normaler Wert oder `Ausnahme`

  ```
  data Maybe a  = Just a | Nothing
  data Either a = Left String | Right a
  ```

- Unscharfe Ausnahmen:

  Wert eines Ausdrucks = Normaler Wert oder Menge von möglichen Ausnahmen

  - Menge wird nicht konstruiert — semantisches Konstrukt.

Unscharfe Ausnahmen fangen.

- Ausnahmen fangen ist monadisch:
  - Funktion `bogus :: a-> (Exception-> a)-> a` hätte alten Probleme
- Determinisierung trennen von Ausnahmebehandlung:
  - `evaluate :: a-> IO a` wertet Ausdruck aus, wirft ggf. mögliche Ausnahme.
  - Ausnahme durch Auswertungsreihenfolge bestimmt.
- `catch :: IO a-> (Exception-> IO a)-> IO a` wie vorher.
- `Unscharfe` Ausnahmen können überall geworfen, aber nur im `IO`-Monaden gefangen werden.
  - Ausnahmen: semantisches `GOTO`.

Asynchrone Ausnahmen

Modelliert durch

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

- Ausnahme wird in anderem Thread geworfen.
- Modelliert alle Situationen wie Interrupts etc.

Asynchrone Ausnahmen: Beispiel

Parallele Auswertung zweier IO-Statements:

- Wer zuerst fertig ist beendet Auswertung.

```
parIO :: IO a-> IO a-> IO a
parIO a1 a2 =
  do m <- newEmptyVar;
     c1 <- forkIO (a1 >>= putMVar m)
     c2 <- forkIO (a2 >>= putMVar m)
     r<- takeMVar m
     throwTo c1 Kill
     throwTo c2 Kill
     return r
```

Asynchrone Ausnahmen: Beispiel

Timeout-Operator:

- Wenn kein Ergbnis nach `n` Mikrosekunden, `Nothing`

```
timeout :: Int-> IO a-> IO (Maybe a)
timeout n a = parIO (r<- a; return (Just r))
                    (threadDelay n; return Nothing)
```

Unscharfe Ausnahmen: Benutzung

- Zur Benutzung: `import Control.Exception` (nur `ghc`)
- Achtung: per default normale Ausnahmen (Haskell98) definiert
  - Überlagerung durch Import, ggf. alte Funktionen verstecken:

```
import Prelude hiding (catch)
import IO hiding (catch,try)
```

- Ausnahmen fangen:

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

Dynamische Typen

Dynamische Typen (`Data.Dynamic`): Repräsentation aller Typen in einem Typen `Dynamic` zur Laufzieit.

```
class Typeable a where
  typeOf :: a -> TypeRep
```

```
toDyn :: (Typeable a) => a -> Dynamic
fromDyn :: (Typeable a) => Dynamic -> a -> a
fromDynamic :: (Typeable a) => Dynamic -> Maybe a
```

In einfachstem Fall (nicht-polymorph, first-order):

```
mkTyCon :: String -> TyCon
mkAppTy :: TyCon -> [TypeRep] -> TypeRep
```

Dynamische Ausnahmen: Beispiel

- Eigene Fehlerwerte definieren:

```
data MyError = Panic String
             | IOFailure Handle String
             | Warning Int String
```

- Als Instanz der Typklasse `Typeable`

```
instance Typeable MyError where
   typeOf _ = mkAppTy (mkTyCon "M.MyError") []
```

- Werfen: `throwDyn (Panic ``HELP!!'')`
- Fangen:

```
case e of DynException d->
  case fromDynamic d of Maybe (Panic str) -> ...
                        Maybe (IOFailure h str) -> ...
```

Ausnahmen: Achtung!

Ausnahmen und Nebenläufigkeit

- Ausnahmen können in anderen Thread geworfen werden!

```
ch <- newChan
forkIO (forever $ do readChan ch >>= putStrLn)
catch (do let x= ...
          writeChan ch x)
      (\e -> ...)
```

- Ausnahme wird in reader-Thread geworfen!
  - Abhilfe: Auswertung mit `evaluate` forcieren.

Ausnahmen: Achtung!

Fehlerabfrage ersetzt keine Ausnahmebehandlung:

```
b <- doesDirectoryExist name
when (not b) $ createDirectory name
```

- Zweite Aktion kann fehlschlagen!

Zusammenfassung

Heute Ausnahmezustand:

- Ausnahmen in Haskell98
- Unscharfe Ausnahmen
- Dynamische Ausnahmen

Nächste Woche: mein bester Freund ist eine C-Funktion, oder: das FFI.

Sprachinteroperabilität

Heute hier:

- Fremde Sprachen aus Haskell aufrufen (call-out)
- Haskell aus fremden Sprachen ausrufen (call-in)
- Datenübertragung zwischen den Grenzen: marshalling
- Das Haskell Foreign Function Interface (FFI)

Grundlagen

- Manuel ‘Chilli’ Chakravarty (ed):
  The Haskell98 Foreign Function Interface 1.0
- GHC User’s Guide, Kapitel 8.
- Simon Peyton-Jones: Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell.

Probleme bei der Sprachinteroperbilität

- Unterschiedliche Datentypen
- Unterschiedliche Auswertungsparadigmen
- Unterschiedliche Parameterübergabedisziplin
  - Call by reference, call by value, im Register, auf dem Stack, Reihenfolge. . .
- Meist sehr ad-hoc gelöst

Haskell und das Standard-FFI

- Spracherweiterung zur Deklaration von
  - importierten Fremdfunktionen
  - exportierten Haskellfunktionen
- Entwurfsphilosophie:
  - Sehr einfach, keine Betonung auf Komfortabilität;
  - Ziel: FFI als standardisiertes back-end für andere Werkzeuge

Haskell und das Standard-FFI

- Spracherweiterung zur Deklaration von
  - importierten Fremdfunktionen
  - exportierten Haskellfunktionen
- Entwurfsphilosophie:
  - Sehr einfach, keine Betonung auf Komfortabilität;
  - Ziel: FFI als standardisiertes back-end für andere Werkzeuge
- Module zum einfachen Marshalling
  - Marshalling: Konversion/Aufbereitung der Daten zwischen den Sprachen
  - Betonung liegt auf Haskell-seitigem Marshalling
- Unterstützt C, C++, JVM, Win32, .NET

Deklaration importierter Fremdfunktionen

- Eine einfache C-Funktion mit

  ```
  foreign import ccall toLower :: Char -> Char
  ```

- C-Funktion mit Seiteneffekten und anderem Namen

  ```
  foreign import "PutChar" putChar :: Char-> IO ()
  ```

- Sicherheit: `safe` vs. `unsafe`
  - `safe` (Default): Funktion kann wieder Haskell aufrufen.
  - `unsafe`: Funktion ruft nicht wieder Haskell auf.
  - Beispiel:

    ```
    foreign import "PutChar" unsafe putChar :: Char-> IO ()
    ```

- Achtung: Der Typ wird nicht überprüft!

Deklaration exportierter Haskellfunktionen

- Exportierte Funktion:

```
foreign export ccall "Foo" foo :: Int-> Int
```

- Export von Instanzen:

```
double :: Num a=> a-> a
foreign export ccall "doubleI"
                     double :: Int-> Int
foreign export ccall "doubleF"
                     double :: Float-> Float
```

Fremdtypen (foreign types)

- Externe Entitäten auf Fremdtypen beschränkt:
- Basisfremdtypen sind:
  - `Char`, `Int`, `Double`, `Float`, `Bool`;
  - `Int8`, ..., `Word64`, `Ptr a`, `StablePrt a`, aus Modul `Foreign`
- Fremdtypen sind:

$$a_1 \texttt{ -> } a_2 \texttt{ -> } \ldots \texttt{ -> } r$$

  - a_i sind Basisfremdtypen, oder Typsynonyme, oder `newtype`'s davon;
  - r dito, oder `()`, oder $IO\ a$ (mit a dito)

Marshalling

- `Int` und `Word`
- Module `Foreign` und `CForeign`
- Abbildung Typen (Haskell) $\longleftrightarrow$ Typen (C)
  - Siehe FFI-Spezifikation, S. 22 (Tabelle 2)

Speicherverwaltung I: `Ptr` und `Storable`

- Typ `Ptr a` abstrakte Adresse auf Objekt vom Typ `a`
  - Instanz von `Eq`, `Ord`, `Show`
- Ausgezeichneter Zeiger ins 'Nichts'

```
nullPtr :: Ptr a
```

- Pointerarithmetik, Alignment, Casting

```
plusPtr  :: Ptr a -> Int-> Ptr a
minusPtr :: Ptr a -> Int-> Ptr a
alignPtr :: Ptr a -> Int-> Ptr a
castPtr  :: Ptr a -> Ptr b
```

Die Klasse `Storable`

- Klasse `Storable` erlaubt Zugriff auf Inhalt der `Ptr`
  - Instanzen: Basisdatentypen, `Int` und `Word`, Zeiger
- Größe und Alignment:

```
sizeOf    :: Storable a=> a -> Int
alignment :: Storable a=> a-> Int
```

- Peek und Poke:

```
peek :: Storable a=> Ptr a-> IO a
poke :: Storable a=> Ptr a-> a-> IO ()
```

  - Alignment beachten!
  - Weitere Hilfsfunktionen für indizierte Zugriffe in Felder

```
peekElemOff :: Storable a=> Ptr a-> Int-> IO a
pokeElemOff :: Storable a=> Ptr a-> Int-> a-> IO ()
```

Memory Management I: `ForeignPtr`

- Das Problem:
  - In imperative Sprache explizite Finalisierung (Resourcefreigabe)
  - Aber in Haskell: implizite *garbage collection*

Memory Management I: `ForeignPtr`

- Das Problem:
  - In imperative Sprache explizite Finalisierung (Resourcefreigabe)
  - Aber in Haskell: implizite *garbage collection*

- Lösung: `ForeignPtr a` mit Finalizer
  - Wird aufgerufen, bevor Objekt abgeräumt wird.

```
data ForeignPtr a
newForeignPtr :: Ptr a-> IO ()-> IO (ForeignPtr a)
```

Memory Management II: `StablePtr`

- Noch einProblem:
  - Imperative Sprachen erwarten unveränderliche Referenzen
  - in Haskell sind Adressen flüchtig ($garbage\ collection$)

Memory Management II: `StablePtr`

- Noch einProblem:
  - Imperative Sprachen erwarten unveränderliche Referenzen
  - in Haskell sind Adressen flüchtig (*garbage collection*)

- Die Lösung: `StablePtr a`
  - Stabile Adresse garantiert.

```
data StablePtr a
newStablePtr :: a-> IO (StablePtr a)
deRefStablePtr :: StablePtr a-> IO a
freeStablePtr :: StablePtr a-> IO ()
```

From Heaven to Hell

Haskell importiert C:

```
module Main where

foreign import ccall "hello.h hello" helloWorld :: IO ()

main =
     do putStrLn "Here you go: "
        helloWorld
```

- Name der C-Quelldatei unerheblich
- Benötigte Deklarationen in `hello.h`
- Übersetzen:

```
ghc -c -ffi Heaven2Hell.hs
gcc -c hello.c
ghc -o hello Heaven2Hell.o hello.o
```

From Hell to Heaven

Haskell-Modul mit exportierter Funktion:

```
module Hell2Heaven where

foreign export ccall "fac" fac :: Int-> Int

fac :: Int-> Int
fac n | n <= 0    = 1
      | otherwise = n* fac (n-1)
```

- Compilation mit `ghc` erzeugt:

  ```
  Hell2Heaven_stub.c -> Hell2Heaven_stub.o
  Hell2Heaven_stub.h
  ```

- Einbinden von `fac` in `callHs.c`:
  - `Hell2Heaven_stub.h` enthält Signatur (Prototypen)
  - Haskell-Laufzeitsystem initialisieren/beenden
  - Alle Haskell-Funktionen in einem Modul zusammengefassen

- Kompilieren:

```
ghc -c -ffi Hell2Heaven.lhs
gcc -I /usr/local/lib/ghc-5.04.3/include \
    -c callHs.c
ghc -no-hs-main -o callHs callHs.o \
      Hell2Heaven.o Hell2Heaven_stub.o
```

  - Pfad auf `ghc`-includes systemspezifisch
  - Linken mit `ghc`, aber `main` aus `callHs.o`

Generell zu beachten

- Typen werden nicht überprüft
- Beliebige Typkonversionen möglich
- Fremdfunktionen können Haskell-Heap korrumpieren

Auf dem FFI aufsetzende Tools

- `GreenCard`
  - Generiert Marshalling aus Direktiven
  - Kein Export
- `C->Haskell`
  - Generiert Marshalling aus Haskell mit Direktiven und C Headerdateien
- `H/Direct`
  - Generiert Marshalling aus IDL
  - Import und Export; unterstützt für Java, C, COM+
  - Groß und etwas unhandlich

Zusammenfassung

- Das Foreign Function Interface (FFI)
  - Standardisiert
  - Einfach gehalten
  - marshalling in Haskell
- Externe Funktionen aus Haskell aufrufen.
- Funktionen aus Haskell heraus exportieren.
- Andere Werkzeuge für das FFI.
- Nächste Woche: Speed!

Effizienzaspekte der funktionalen Programmierung

Heute! Hier!

Wie geht's nach Weihnachten weiter?

- Fallbeispiel: der Haskell Webserver.
- Grafik: `hOpenGL`
- GUIs: `wxHaskell`, `HTk`
- Fancy Types:
  - In Haskell: Existentielle Typen, Rang-2-Polymorphie, extendable records;
  - In anderen Sprachen: abhängige Typen.

Effizienzaspekte

- Beste Lösung: bessere Algorithmen.
- Zweitbeste Lösung: Büchereien nutzen.
- Effizenzverbesserungen durch
  - Endrekursion und Striktheit: Speicherlecks vermeiden
  - Der ewige Konflikt: Geschwindigkeit vs. Platz

Endrekursion

Eine Funktion ist endrekursiv, wenn kein rekursiver Aufruf in einem geschachtelten Ausdruck steht.

- D.h. darüber nur `if`, `case`, guards oder Fallunterscheidungen.
- Entspricht `goto` oder `while` in imperativen Sprachen.
- Wird in Schleifen übersetzt.
- Nicht-endrekursive Funktionen brauchen Platz auf dem Stack.

Beispiele

- `fac'` nicht endrekursiv:

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

- `fac` endrekursiv:

```
fac :: Integer-> Integer
fac n      = fac0 n 1 where
  fac0 n acc = if n == 0 then acc
                         else fac0 (n-1) (n*acc)
```

- `fac'` verbraucht Stackplatz, `fac` nicht. (Zeigen)

Beispiele

- Liste umdrehen, nicht endrekursiv:

```
rev' :: [a]-> [a]
rev' []     = []
rev' (x:xs) = rev' xs ++ [x]
```

  - Hängt auch noch hinten an — $O(n^2)$!

Beispiele

- Liste umdrehen, nicht endrekursiv:

```
rev' :: [a]-> [a]
rev' []     = []
rev' (x:xs) = rev' xs ++ [x]
```

  - Hängt auch noch hinten an — $O(n^2)$!

- Liste umdrehen, endrekursiv und $O(n)$: (Zeigen)

```
rev :: [a]-> [a]
rev xs = rev0 xs [] where
    rev0 []     ys = ys
    rev0 (x:xs) ys = rev0 xs (x:ys)
```

Überführung in Endrekursion

- Gegeben eine Funktion f' `f' :: S -> T`

```
f' x = if B x  then H x
               else φ (f' (K x)) (E x)
```

  - Mit `K :: S -> S`, `φ :: T -> T -> T`, `E :: S -> T`, `H :: S -> T`.

- Sei ϕ assoziativ ist, e `:: T` neutrales Element
- Dann ist die endrekursive Form:

```
f :: S -> T
f x = g x e where
 g x y = if B x  then φ (H x) y
                 else g (K x) (φ (E x) y)
```

Beispiel

- Länge einer Liste

```
length' :: [a]-> Int
length' xs = if (null xs) then 0
             else 1+ length' (tail xs)
```

- Zuordnung der Variablen:

$$\begin{array}{rcl} K(x) & \mapsto & \texttt{tail} \\ E(x) & \mapsto & 1 \\ \phi(x,y) & \mapsto & x+y \end{array} \qquad \begin{array}{rcl} B(x) & \mapsto & \texttt{null}x \\ H(x) & \mapsto & 0 \\ e & \mapsto & 0 \end{array}$$

- Es gilt: $\phi(x,e) = x + 0 = x$ (0 neutrales Element)

- Damit ergibt sich endrekursive Variante:

```
length :: [a]-> Int
length xs = len xs 0 where
  len xs y = if (null xs) then 0 -- was: 0+ 0
             else len (tail xs) (1+ y)
```

- Allgemeines Muster:
  - Monoid (ϕ, e): ϕ assoziativ, e neutrales Element.
  - Zustäzlicher Parameter akkumuliert Resultat.

Endrekursive Aktionen

Eine Aktion ist endrekursiv, wenn nach dem rekursiven Aufruf keine weiteren Aktionen folgen.

- Nicht endrekursiv:

```
getLines' :: IO String
getLines' = do str<- getLine
               if null str then return ""
                  else  do rest<- getLines'
                           return (str++ rest)
```

- Endrekursiv:

```
getLines :: IO String
getLines = getit "" where
   getit res = do str<- getLine
                  if null str then return res
                     else getit (res++ str)
```

Fortgeschrittene Endrekursion

- Akkumulation von Ergebniswerten durch closures
  - closure: partiell instantiierte Funktion
- Beispiel: die Klasse `Show`
  - Nur `show` wäre zu langsam ($O(n^2)$):

```
class Show a where
   show :: a-> String
```

Fortgeschrittene Endrekursion

- Akkumulation von Ergebniswerten durch closures
  - closure: partiell instantiierte Funktion
- Beispiel: die Klasse `Show`
  - Nur `show` wäre zu langsam ($O(n^2)$):

```
class Show a where
    show :: a-> String
```

  - Deshalb zusätzlich

```
showsPrec :: Int-> a-> String-> String
show x    = showsPrec 0 x ""
```

  - String wird erst aufgebaut, wenn er ausgewertet wird ($O(n)$).

- Damit zum Beispiel:

```
data Set a = Set [a] -- Mengen als Listen

instance Show a=> Show (Set a) where
  showsPrec i (Set elems) =
      \r-> r++ "{" ++ concat (intersperse ", "
                              (map show elems)) ++ "}"
```

- Damit zum Beispiel:

```
data Set a = Set [a] -- Mengen als Listen

instance Show a=> Show (Set a) where
  showsPrec i (Set elems) =
      \r-> r++ "{" ++ concat (intersperse ", "
                              (map show elems)) ++ "}"
```

- Nicht perfekt— besser:

```
instance Show a=> Show (Set a) where
  showsPrec i (Set elems) = showElems elems where
    showElems []     = ("{}" ++)
    showElems (x:xs) = ('{' :) . shows x . showl xs
       where showl []     = ('}' :)
             showl (x:xs) = (',' :) . shows x . showl xs
```

Speicherlecks

- Garbage collection gibt unbenutzten Speicher wieder frei.
  - Nicht mehr benutzt: Bezeichner nicht mehr im Scope.
- Eine Funktion hat ein Speicherleck, wenn Speicher belegt wird, der nicht mehr benötigt wird.
- Beispiel: `getLines`, `fac`
  - Zwischenergebnisse werden nicht auswertet. (Zeigen.)
  - Insbesondere ärgerlich bei nicht-terminierenden Funktionen.

Striktheit

- Funktion `f` ist strikt in einem Argument `x` gdw.

$$\texttt{x} \equiv \bot \Longrightarrow \texttt{f(x)} \equiv \bot$$

- Beispiele:
  - `(+)` strikt in beiden Argumenten.
  - `(&&)` strikt im ersten, nicht-strikt im zweiten.
    `False && error` $\rightsquigarrow$ `False`
  - `silly` nicht-strikt im ersten. `silly error 3` $\rightsquigarrow$ `3`
  - Zeigen.
- Haskell ist
  - als nicht-strikt spezifiziert;
  - (meist) mit verzögerter Auswertung implementiert.

Striktheit

- Strikte Argumente erlauben Auswertung vor Aufruf
  - Dadurch konstanter Platz bei Endrekursion.
- Striktheit durch erzwungene Auswertung:
  - `seq :: a-> b-> b` wertet erstes Argument aus.
  - `($!) :: (a-> b)-> a-> b` strikte Funktionsanwendung

```
f $! x = x `seq` f x
```

- Fakultät in konstantem Platzaufwand (zeigen):

```
fac2 n = fac0 n 1 where
  fac0 n acc = seq acc $ if n == 0 then acc
                         else fac0 (n-1) (n*acc)
```

`foldr` vs. `foldl`

- `foldr` ist nicht endrekursiv.
- `foldl` ist endrekursiv:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- `foldl' :: (a-> b-> a)-> a-> [b]-> a` ist endrekursiv und strikt.

- `foldl` endrekursiv, aber traversiert immer die ganze Liste.
- `foldl'` konstanter Platzaufwand, aber traversiert immer die ganze Liste.
- Wann welches `fold`?
  - Strikte Funktionen mit `foldl'` falten.
  - Wenn nicht die ganze Liste benötigt wird, `foldr`:

```
all :: (a-> Bool)-> [a]-> Bool
all p = foldr ((&&) . p) True
```

Gemeinsame Teilausdrücke

- Ausdrücke werden intern durch Termgraphen dargestellt.
- Argument wird nie mehr als einmal ausgewertet:
  - `trace :: String-> a-> a` druckt `String` bei Auswertung ((ghc) (Zeigen) `import IOExts (trace)`

```
f :: Int-> Int-> Int
f x y = x+ x
test1 = f (trace "Eins\n" (3+2))
          (trace "Zwei\n" (3+2))
```

- Sharing von Teilausdrücken
  - Explizit mit `where` und `let`
  - Implizit (ghc) (Beispiel)

Memoisation

- Fibonacci-Zahlen als Strom: linearer Aufwand.

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- Kleine Änderung der Fibonacci-Zahlen als Strom:

```
fibsFn :: () -> [Integer]
fibsFn () = 1 : 1 : zipWith (+) (fibsFn ())
                                (tail (fibsFn ()))
```

- Große Wirkung: Exponentiell. Warum?

Memoisation

- Fibonacci-Zahlen als Strom: linearer Aufwand.

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- Kleine Änderung der Fibonacci-Zahlen als Strom:

```
fibsFn :: () -> [Integer]
fibsFn () = 1 : 1 : zipWith (+) (fibsFn ())
                                (tail (fibsFn ()))
```

- Große Wirkung: Exponentiell. Warum?
  - Jeder Aufruf von `fibsFn()` bewirkt erneute Berechnung.

Memoisation

  - Die Abhilfe: Memoisation

- Memoisation: Bereits berechnete Ergebnisse speichern.
- In Hugs: Aus dem Modul `Memo`:

```
memo :: (a-> b)-> a-> b
```

- Damit ist alles wieder gut (oder?)

```
fibsFn' :: () -> [Integer]
fibsFn' = memo (\()-> 1 : 1 : zipWith (+)
                                (fibsFn' ())
                                (tail (fibsFn' ())))
```

- GHC kann das automatisch.

Überladene Funktionen sind langsam.

- Typklassen sind elegant aber langsam.
  - Implementierung von Typklassen: dictionaries von Klassenfunktionen.
  - Überladung wird zur Laufzeit aufgelöst.
- Bei kritischen Funktionen durch Angabe der Signatur Spezialisierung erzwingen.
- NB: Zahlen (numerische Literale) sind in Haskell überladen!
  - Bsp: `facs` hat den Typ `Num a=> a-> a`

```
facs n = if n == 0 then 1 else n* facs (n-1)
```

Listen als Performance-Falle

- Listen sind keine Felder.
- Listen:
  - Beliebig lang
  - Zugriff auf n-tes Element in linearer Zeit.
  - Abstrakt: frei erzeugter Datentyp aus Kopf und Rest
- Felder:
  - Feste Länge
  - Zugriff auf n-tes Element in konstanter Zeit.
  - Abstrakt: Abbildung Index auf Daten

- Modul `Array` aus der Standardbücherei

```
data Ix a=> Array a b  -- abstract
array     :: (Ix a) => (a,a) -> [(a,b)]
                                -> Array a b
listArray :: (Ix a) => (a,a) -> [b] -> Array a b
(!)       :: (Ix a) => Array a b -> a -> b
(//)      :: (Ix a) => Array a b -> [(a,b)]
                                -> Array a b
```

- Als Indexbereich geeignete Typen (Klasse `Ix`): `Int`, `Integer`, `Char`, `Bool`, Tupel davon, Aufzählungstypen.
- In Hugs/GHC vordefiniert (als "primitiver" Datentyp)

Listen sind keine `FiniteMaps`!

- `FiniteMap`: Assoziationslisten oder endliche Abbildungen
  - Effizient, weil durch balancierte Bäume implementiert.

```
data Ord key => FiniteMap key elt
```

- Alles fängt mit `emptyFM` an.
- Hinzufügen mit `addToFM`, `addToFM_C`.
  - Verhalten bei Überschreiben kann spezifiziert werden.
- Auslesen mit `lookupToFM`, `lookupWithDefaultFM`.

Unboxed Types

- boxed type: Elemente Objekte auf dem Heap
  - Vgl. Wrapper-Klassen in `Java`
- unboxed type: direkte Repräsentation als Maschinenworte
  - z.B. `Int#` (`long int`), `Double#` (`double`), `Addr#` (`void *`);
  - Operationen z.B. `(+#)` entspricht Addition auf Maschinenebene.
  - Objekte nterliegen nicht der garbage collection.
- Verschiedene Restriktionen (nicht für polymorphe Funktionen etc.)
- Nützlich: für numerik-intensive Berechnungen, innere Schleifen.

Zusammenfassung

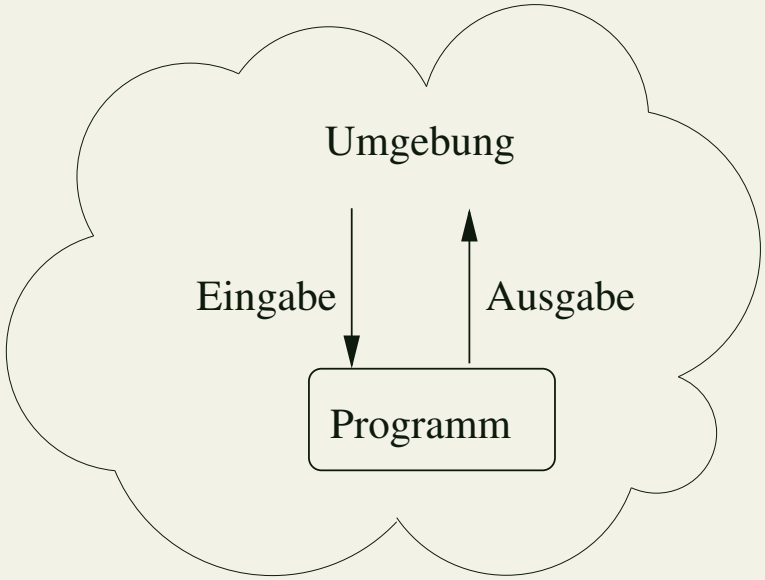
- Endrekursion: `while` für Haskell.
  - Überführung in Endrekursion meist möglich.
  - Noch besser sind strikte Funktionen.
- Speicherlecks vermeiden: Striktheit, Endrekursion und Memoisation.
- Überladene Funktionen sind langsam.
- Listen sind keine Arrays.
- Referenzen in Haskell: `IORef`

GUIs und reaktive Programmierung

Und heute wieder. . .

- Reaktive Programmierung
  - Abschied vom sequentiellen 1–2–3.
- Programmierung grafischer Benutzerschnittstellen (GUIS) in Haskell:
  - HTk — Tcl/Tk in Haskell
  - wxHaskell — wxWidgets in Haskell

Reaktive Programmierung



Sequentiell

Reaktiv

Beispiele für reaktive Programme: GUIs, Applets, eingebettete Systeme

Reaktive Programme

Herkömmliches sequentielles Programmiermodell basiert auf

- Sequentielle Zustandsübergänge
- Ein/Ausgabe durch Seiteneffekte

Hier unzureichend:

- Grundlegende Operationen: auf Eingabe warten, Ausgabe senden
- Mehrere Eingaben gleichzeitig, Reihenfolge der Ausgabe nicht spezifiert, Nebenläufigkeit

Problem: Realisierung reaktiver Programme in sequentieller Sprache

Realisierung reaktiver Programme

- Eventschleife im Program (X Windows)
- Externe Eventschleife:
  - Callbacks (Tcl/Tk)
  - Signale (Qt, GTK+)
  - Event handlers (Java: Applets, Swing, AWT)
- Komponierbare Events (HTk)

Zentrale Eventschleife: Xlib

```
#include <X11/Xlib.h>                    #include <X11/Xutil.h>
int main(int argc, char **argv) {
 Display *dply; XEvent event;
 /* draw some graphics here */
 for (;;) {   XNextEvent(dply, &event);
   if (event.type == ButtonPress) {
       printf("Button pressed at (%d, %d)\n",
              event.xbutton.x, event.xbutton.y);
     } else if (event.type == KeyPress) {
       KeySym ks;
       ks= XLookupKeysym(&event.xkey, 0);
       printf("Key %04x (%s) pressed.\n", ks, XKeysymToString(
   XFlush(dply); }}
```

Zentrale Eventschleife

Vorteile:

- Direkte Kontrolle
- Schnelligkeit

Nachteile:

- umständlich, fehleranfällig
- Nicht kompositional

Eventbehandlung mit Callbacks, Signals etc

- Eine Eventschleife außerhalb des reaktiven Programms
- Programm implementiert Prozeduren, die Events verarbeiten.
- Prozedurparameter: Event-Daten, zum Beispiel gedrückte Taste oder Mausknopf; Zusatzinformationen wie Mausposition.
- Im Beispiel oben:

```
typedef struct {int button; int x_pos; int y_pos} mouse_info

void handle_buttonPress(mouse_info i);
void handle_keyPress(char key);
```

- Unterschiede: Art der Bindungsbehandlung

Callbacks in Tcl/Tk

- Tcl/Tk: *user interface library and programming language*
- Tcl (tool command language): String-basierte Kommandosprache
- Tk (graphical toolkit): Bibliothek von Funktionen zum Erstellen von GUIs
- Philosophie: Leichtgewichtige Skriptsprache, schnelle Entwicklung von GUIs durch das Zusammenkleben von Applikationen (glueware)
- Plattformunabhängig: X, Windows, MacOS, . . .
- Kein native look&feel

Callbacks in Tcl/Tk

- Bindung von Kommandos an Events durch `bind`:

```
button .b -text Test
bind .b <Enter> {puts "Button Entered!"}
bind .b <ButtonPress-2> { puts "Button pressed at (%x, %y)
pack .b
```

- Völlig ungetypt:
  - Funktionen höherer Ordnung durch Quoting
  - Implizite Parameterübergabe über %-Substitution
- Ablauf:
  - Aufbau des Interface
  - Festlegung des dynamischen Verhaltens
  - Interface darstellen und externe Eventschleife aufrufen

Event Handling in Java

- Eventverabeitende Methoden leben in Unterklassen der `Listener`-Klassen:
  - `interface ActionListener` für Aktionen (z.B. Knopfdrücke);
  - `interface MouseListener` für Mouse-Events.
- In `Listener` werden Event-bearbeitende Methoden überladen:

```
interface ActionListener {
 public void actionPerformed(ActionEvent e)
 }
```

- Listener an Event-Quellen binden:

  `addActionListener(Object o)`, `addMouseListener(Object o)`

Beispiel: ein einfaches Applet

```
public class Beeper extends JApplet
                    implements ActionListener {
    JButton button;
    public void init() {
        button = new JButton("Click Me");
        getContentPane().add(button, BorderLayout.CENTER);
        button.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }
}
```

Zusammenfassung: Callbacks und Event Handling

Vorteile:

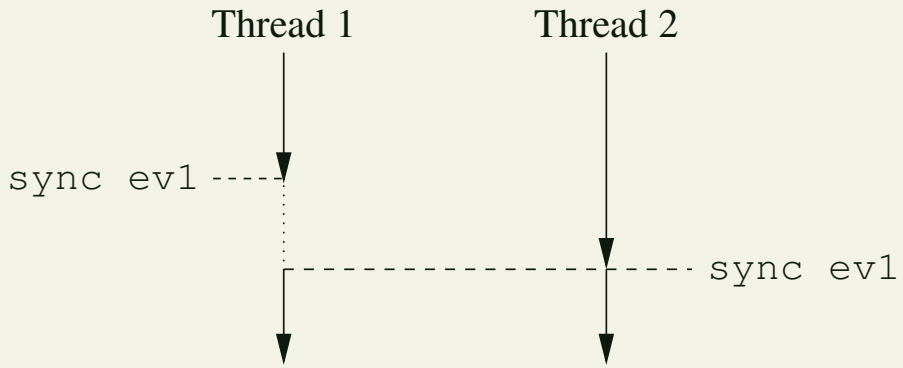
- abstrakter als explizite Eventschleife
- kompositional innerhalb des Toolkits (e.g. zwei Swing-Programme)

Nachteile:

- Mangelnde Modularität und Kompositionalität, weil es immer nur eine Eventschleife geben kann
- Zustandsbasiert

Funktionale reaktive Programmierung: `Events`

- Eine grundlegende Operation: Synchronisation



- Beschreibung der Synchronisationspunkte: `Event a`
- Kombination von Events mit `+>` (Auswahl)

```
type Event a
sync    :: Event a -> IO a
(+>)    :: Event a -> Event a -> Event a
(>>>=) :: Event a -> (a-> IO b) -> Event b
```

Abgeleitete Events

- Mit `always :: Event a` kann immer synchronisiert werden.
- Events als Monaden:

```
instance Monad Event where
  (>>=) ev1 getEv2 = ev1 >>>= sync . getEv2
  return v         = always (return v)

-- Damit z.B:
forever :: Event a-> Event ()
forever e = do e; forever e

spawnEvent :: Event a-> IO (IO ())
spawnEvent r = do i<- forkIO (sync r); return (killThread i)
```

Beispiel: Events in HTk

```
main :: IO ()
main =
  do htk <- initHTk [text "Main Window"]

     b <- newButton htk [text " Quit ", size(10, 2)]
     pack b []

     press <- clicked b

     spawnEvent (forever (press >>> (putStrLn "Exiting!" >> de

     finishHTk
```

HTk

- HTk: graphische Benutzerschnittstellen für Haskell
- Eigenentwicklung Uni Bremen
- Benutzt Tk, aber ersetzt Tcl durch Haskell
- Voll nebenläufig
- Getypte, komponierbare Events als „first-class objects“
- Deckt den gesamten Funktionsumfang von Tk ab:
  widgets, text widgets, canvasses, . . .

Beispiele in HTk

- htkSimple1.hs
- htkSimple2.hs
- hsMines.hs
- hsMines2.hs

wxHaskell

- Basiert auf wxWidgets
  - “a single, easy-to-use API for writing GUI applications on multiple platforms.”
  - Implementiert auf Linux/GTk2, Windows, Mac OS X, . . .
- wxHaskell ist Haskell-Bindung
- Event Handling (externe Eventschleife á la Tcl, Java etc.)
- Vielseitig, aber groß (40 MB zzgl. wxWidgets)

Beispiele in wxHaskell

- HellowWx.hs
- BouncingBall.hs

Zusammenfassung

- Reaktive Programme mit traditionellen sequentiellen Programmiermodell schwer zu fassen — Ansätze:
  - Interne Eventschleifen (Programm durchläuft alle Eingaben)
  - Externe Eventschleifen (Programm als Sammlung von Funktionen, die Eingaben behandeln)
  - Nachteile: Kompositionalität und Modularität
- Funktionale Sprachen erlauben abstraktere Ansätze:
  - Ereignisbasierte Programmierung (HTk)
  - Vorteile HTk: kompositional und modular, nebenläufig
    Nachteile HTk: etwas angestaubt, kein native look&feel
  - Vorteile wxHaskell: größerer Funktionsumfang,
    Nachteil wxHaskell: externe Eventschleife, nicht voll nebenläufig

Fallstudie: der Haskell Web Server

Unser Tagesmenü:

- Der Haskell Web Server — eine Fallstudie in
  - reaktiver,
  - nebenläufiger,
  - hochverfügbarer Programmierung.
- Simon Marlow: Writing High-Performance Server Applications, Haskell Workshop 2000.
- Disclaimer: übersetzt nicht, da für `ghc-5.x`
  - $\Longrightarrow$ paradigmatisch.

Warum ein Web Server in Haskell?

- Einfaches, aber nicht triviales Protokoll — `HTTP`
- Nebenläufigkeit
  - viele Requests gleichzeitig
- Geringe Latenz
  - schnell Antworten
- Fehlertoleranz und Verfügbarkeit
  - Fehler im Server, Fehler auf Clientseite, Konfigurierbarkeit

Nebenläufigkeitsmodell

- Betriebssystemprozesse
  - Einfach, mehrprozessorfähig; aber: schwergewichtig
- Betriebssystemthreads
  - Einface IPC, mehrprozessorfähig; aber: schwierig;
- Ein Prozess (`select()` oder asynchrone I/O)
  - schnell; aber: mangelnde Abstraktion
- Benutzerthreads (Concurrent Haskell)
  - Einfach, leichtgewichtig, schnell; aber: nicht mehrprozessorfähig.

Struktur

Top-Level Schleife:

```
acceptConnections conf ps sock = do
(h, SockAddrInet port haddr) <- accept sock
forkIO ( (talk conf ps h haddr  `finally`  (hClose h))
          `catch`
         (\e -> trace ("servlet died: "  ++ show e) (return
       )
acceptConnections conf ps sock
```

- Aufräumen mit `finally`

Serving Requests

1. Request lesen
2. Request parsieren
3. Antwort generieren
4. Antwort senden
5. Wenn Verbindung aufrecht erhalten ($keep\ alive$), dann zurück zu 1.

Request lesen

```
getRequest :: Handle -> IO [String]
getRequest h = do
  l <- hGetLine h
  if (emptyLine l) then getRequest h else getRequest' l h

getRequest' l h = do
  if (emptyLine l) then return []
     else do l' <- hGetLine h; ls <- getRequest' l' h
             return (l:ls)
```

Request parsieren

- Konkrete Syntax:

```
GET /~cxl/ HTTP/1.1
Host: kierkegaard.informatik.uni-bremen.de
Date: Wed Jan 12 16:20:11 CET 2005
```

- Abstrakte Repräsentation von Requests:

```
data Request = Request {
    reqCmd      :: RequestCmd,
    reqURI      :: ReqURI,
    reqHTTPVer  :: HTTPVersion,
    reqHeaders  :: [RequestHeader],
    reqFilename:: FilePath }
data RequestCmd  = OptionsReq | GetReq | HeadReq ...
```

Repräsentation der Antworten

```
data ResponseBody
  = NoBody | FileBody Integer{-size-} FilePath | HereItIs Stri

data Response = Response {
  respCode     :: ResponseCode,
  respHeaders  :: [String],
  respCoding   :: [TransferCoding],
  respBody     :: ResponseBody,
  respSendBody :: Bool,
  respContentType :: Maybe String  }
```

Antworten generieren

- Der Normallfall: eine Datei senden (“servieren”)
  - Neuere Version hat plug-in Architektur $\Longrightarrow$ “Servlets”
- Achtung: Datei nicht erst in `String` konvertieren.

- Datei senden:

```
bufsize = 4 * 1024 :: Int
squirt :: Fd-> Fd-> IO ()
squirt rd wr = do
  arr <- newArray (0, bufsize-1)
  let loop = do r <- hGetArray rd arr bufsize
                if (r == 0) then return ()
                   else if (r < bufsize)
                           then hPutArray wr arr r
                           else hPutArray wr arr bufsize >>
  loop
```

Timeouts

- Wichtig: wenn keine Antwort kommt, Verbindung abbrechen.
- Generischer Timeout-Kombinator:

```
timeout :: Int       -- timeout in Sekunden
        -> IO a      -- normale Aktion
        -> IO a      -- timeout-Aktion
        -> IO a
```

Rekonfigurierbarkeit

- Wichtig: zur Laufzeit rekonfigurierbar
- Dafür: asynchrone Ausnahmen

```
topServer conf ps = do
    catchError (server conf ps)
        (\e -> case e of
                   ErrorCall "**restart**" -> do readConfig
server = ...
```

-

```
readConfig = do
    blockSignals sigsToBlock   --- Signale blockieren!
    r <- parseConfig configPath
    case r of
      Right b  -> do ...
        topServer conf' plugins
```

Server main

```
main :: IO ()
main = do
  main_thread <- myThreadId
  installHandler sigPIPE Ignore Nothing
  installHandler sigHUP (Catch (hupHandler main_thread)) Nothi
  block $ readConfig

hupHandler :: ThreadId -> IO ()
hupHandler main_thread
  = throwTo main_thread (ErrorCall "**restart**")
```

Performance

- Kleinere Verbesserungen an `ghc` RTS und Büchereien
  - Binär-IO
  - Context-Switches
  - Verbesserung an `hGetLine`
- Latenz: linear ca. 710 Req/s, danach Einbruch.
- Zusammenfassend:
  - Zufriedenstellende Performance
  - . . . in nur ca. 3000 Zeilen Haskell (ursprüngl. 1500)
  - Plugin-Architektur — zur Laufzeit neue Handler hinzufügen.

Mehr Haskell im Web

- `HaskellCGI` — CGI-Skripte in Haskell
  - Direkte Repräsentation von `HTML` — bleibt im WebServer undurchsichtig (bis auf Fehlermeldungen)
  - Nur sehr einfaches Request — Response
- `WASH` — Server-Side Scripting in Haskell
  - Setzt auf `CGI` auf
  - Session-Konzept

Zusammenfassung

- Implementierung eines standardkonformen, stabilen HTTP-Servers in Haskell in ca. 1500 loc möglich;
- Wichtig:
  - Nebenläufigkeit,
  - Stabilität,
  - Verfügbarkeit.
- Darauf aufsetzend Plug-In Rahmenwerk (Servlets)
- Problem: Dokumentation, mangelnde Wartung
- . . . und nächste Woche ?

Domain Specific Languages and Arrows

Heute Hier

- Arrows!
- Was ist das?
- Wozu ist das?
- Was ist eine Domain-Specific Language?

Domain Specific Languages (DSL)

- DSL: Sprache für speziellen Problembereich
  - Im Gegensatz zu universalen Programmiersprachen
  - Beispiel: LaTeX, Shell-Skripte, Spreadsheets, Emacs Lisp, . . .
- Implementierung von DSLs:
  - Einbettung in Haskell

Beispiel: Parserkombinatoren

- Typ für Parser:

```
type Parser a b = [a]-> [(b, [a])]
```

- Monadeninstanz:

```
instance Monad (Parser a) where
  return a = \inp-> [(a, inp)]
  f >>= g  = \inp-> concat [g v out | (v, out)<- f inp]
```

- Problem: Alternierung zu kostspielig.

```
infixl 3 `alt`
alt :: Parse a b-> Parse a b-> Parse a b
alt p1 p2 i = p1 i ++ p2 i
```

Vorausschauend Parsieren

- Lösung: LL(1) Parser mit lookahead
  - Können an ersten Token erkennen, ob es parsiert oder nicht.
  - Konstruktion einer Lookahead-Tabelle

```
data Lookahead  s = L  Bool [s]
data DynamicP s a = DP ([s]-> Maybe (a, [s]))
data Parser s a   = P (Lookahead s) (DynamicP s a)
```

- Kombinatorfunktionen, e.g.

```
token :: s-> Parser s s
token s = P (L False [s]) (DP (\ (x:xs)-> Just (s, xs))
```

  - Kein Test auf leere Eingabe oder `x == xs` nötig
- Damit effiziente Alternierung (Klasse `MonadPlus`):

```
instance MonadPlus Parser where
  P (L e1 s1) (DP p1) `mplus`
  P (L e2 s2) (DP p2) =
    P (L (e1 || e2) (s1 ++ s2)
      (DP (\ x-> case x of
         [] -> if e1 then p1 [] else
               if e2 then p2 [] else Nothing
         (x:xs) -> if x `elem` s1 then p1 (x:xs) else
                   if x `elem` s2 then p2 (x:xs) else
                   if e1 then p1 (x:xs) else
                   if e2 then p2 (x:xs) else Nothing)))
```

- Problem: Kein Instanz mehr für Monad:

  ```
  (>>=) :: Parser s a -> (a-> Parser s b)-> Parser s b
  ```

  - Zweiter Parser nicht mehr statisch konstruierbar.

Arrows!

- Idee: `Arrow a=> a b c` ist Berechnung mit Eingabe `b` und Ausgabe `c`
  - Internalisierung des Funktionstyps `b-> c`.

```
class Arrow a where
  arr :: (b -> c) -> a b c
  pure :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b, d) (c, d)
  second :: a b c -> a (d, b) (d, c)
  (***) :: a b c -> a b' c' -> a (b, b') (c, c')
  (&&&) :: a b c -> a b c' -> a b (c, c')
```

- `arr` oder `pure`: Einbettung von Berechnungen
- `>>>`: Komposition
- `first`, `second`: Berchnung auf erstes (zweites) Argument anwenden.

```
second f = arr flip >>> first f >>> arr flip
```

- `***`: Beide Berechnungen anwenden.

```
f *** g = first f >>> second g
```

- `&&&`: Eingabe auf Berechnungen aufteilen

```
f &&& g = arr (\ b-> (b, b) >>> (f *** g))
```

Lifting

- `Lifting` (auf b): Funktion `a-> b` zu Funktion auf dem Datentypen `d a-> d b`
- Beispiel: monadische Addition:

```
addM :: Monad m=> m Int-> m Int-> m Int
addM a b = do x<- a; y<- b; return (x+ y)
```

- Allgemein:

```
liftM2 :: Monad m=> (a-> b-> c)-> m a-> m b-> m c
liftM2 f a b = do x<- a; y<- b; return (f x y)
```

- Auch für Arrows:

```
liftA2 :: Arrow a=> (b-> c-> d)-> a e b-> a e c-> a e d
liftA2 op f g = (f &&& g) >>> arr (\ (b, c)-> op b c)
```

Parserkombinatoren als Arrows

- Brauchen extra Parameter — Eingabetyp.
  - Lookahead nicht mehr von Eingaben zur Parsierzeit abhänging.

```
data DynamicP s a b = DP ((a, [s])-> Maybe (b, [s]))
data Parser   s a b = P (Lookahead s) (DynamicP s a b)
```

- Damit Instanz wie folgt:

```
instance Arrow (Parser s) where
  arr f = P (L True []) (DP (\ (b, s)-> Just (f b, s)))
  P (L e1 s1) (DP p1) >>> P (L e2 s2) (DP p2) =
    P (L (e1 || e2) (s1 `union` if e1 then s2 else []))
      (DP (\ (b,s)-> case p1 (b, s) of
                       Just (c, s')-> p2 (c, s')
                       Nothing     -> Nothing))
  first (P l (DP p)) =
    P l (\ ((b, d), s)-> case p (b, s) of
                           Just (c, s')-> Just ((c, d), s')
                           Nothing     -> Nothing)
```

Streams als Arrows

- Beispiel: Ströme als Berechnungen
- `SP a b` ist ein *stream processor* von `a` nach `b` mit

```
put :: b-> SP a b-> SP a b     -- b ausgeben, dann weiter
get :: (a-> SP a b)-> SP a b   -- a konsumieren, dann weiter
```

- Mögl. Implementation:

```
data SP a b = Put b (SP a b) | Get (a-> SP a b)
put = Put
get = Get
```

- Damit Instanz für `Arrow`

```
instance Arrow SP where
  arr f = Get (\ (x-> Put (f x) (arr f))
  sp1 >>> Put c sp2   = Put c (sp1 >>> sp2)
  Put b sp1 >>> Get f = sp1 >>> f b
  Get f1 >>> Get f2   = Get (\a-> f1 a >>> Get f2)
```

- Für `first`:

```
first f = bypass [] [] f
bypass [] ds (Get f) = Get (\ (b, d)-> bypass [] (ds++ [d]
bypass (b:bs) [] (Get f) = bypass bs [] (f b)
bypass bs [] (Put c sp)  =
  Get (\ (b, d)-> Put (c, d) (bypass bs++ [d]) [] sp)
bypass [] (b:bs) (Put c sp) =
  Put (c, d) (bypass [] ds sp)
```

  - Ungelesene Eingaben puffern, bis Ausgaben produziert werden
  - Argumente für `bypass`: ungelesene Eingaben (mind. eine leer)

- Für `first`:

```
first f = bypass [] [] f
bypass [] ds (Get f) = Get (\ (b, d)-> bypass [] (ds++ [d]
bypass (b:bs) [] (Get f) = bypass bs [] (f b)
bypass bs [] (Put c sp)  =
  Get (\ (b, d)-> Put (c, d) (bypass bs++ [d]) [] sp)
bypass [] (b:bs) (Put c sp) =
  Put (c, d) (bypass [] ds sp)
```

  - Ungelesene Eingaben puffern, bis Ausgaben produziert werden
  - Argumente für `bypass`: ungelesene Eingaben (mind. eine leer)

- Damit zum Beispiel Fibonacci-Zahlen:

```
fibs  = put 0 fibs'
fibs' = put 1 (liftA2 (+) fibs fibs')
```

Arrow Laws

Folgende Gesetze gelten für `Arrow`:

- Assoziativität
- `id` ist rechts und links neutrales Element
- Funktorialität für `left`, `right`

Weitere Operationen

- Fallunterscheidungen:

```
class (Arrow a) => ArrowChoice a where
  left :: a b c -> a (Either b d) (Either c d)
  right :: a b c -> a (Either d b) (Either d c)
  (+++) :: a b c -> a b' c' -> a (Either b b') (Either c c')
  (|||) :: a b d -> a c d -> a (Either b c) d
```

- Feedback (Rekursion)

```
class (Arrow a) => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c
```

Zusammenfassung

- Arrows sind Verallgemeinerung von Monaden
- 'Internalisierung' des Funktionstyps $\Rightarrow$ besonders nützlich für DSLs.
- Exotischer als Monaden
  - Monaden sind unerläßlich, Arrows sind optional
- Nützlich?

Fancy Types I: Existentielle Typen und Polymorphie höher Ordnung

Heute

- “Sexy Types”:
- Existentielle Typen
  - Objektorientierung in Haskell?
- Polymorphie Höherer Ordnung
  - Typentheorie
- Entscheidbarkeit der Typüberprüfung

Funktional vs. Objektorientiert

- Gemeinsame Konzepte:
  - Polymorphie
  - Verkapselung (Klassen vs. ADTs/Module)
  - Klassenhierarchie
  - abstrakte Klassen (`interface/class`) vs. Instanzen (`class/instance`)
- nur OO: Zustandsbasiertheit, *late binding*
- nur Funktional: algebraische Typen, Fkt. höherer Ordnung. strengere Typisierung

Dynamische Bindung

- Java: Auflösung der Methoden zur Laufzeit
- Hier ist ein Beispiel:

```
class Shape
  { void draw() { System.out.println("Shape"); }}
class Triangle extends Shape
  { void draw() { System.out.println("Triangle"); }}
class Square extends Shape
  { void draw() { System.out.println("Square"); }}
class test {
  public static void main(String [] a)
  ...
  s.draw();
  }
```

Späte Bindung in Haskell?

- Die Klassenhierarchie:

```
class Shape s where draw :: s-> String
                    draw _ = "Shape"
class Shape s=> Triangle s
class Shape s=> Circle s

data TriangleT = Triangle
instance Shape TriangleT where draw Triangle = "Triangle"
instance Triangle TriangleT

data CircleT   = Circle
instance Shape CircleT where   draw Circle   = "Circle"
instance Circle CircleT
```

- aber . . . :

```
main :: IO ()
main = do
  a<- getArgs
  let s= case (head a) of
           "t" -> Triangle; "c"-> Circle
  putStrLn (draw s)
```

- aber . . . :

```
main :: IO ()
main = do
  a<- getArgs
  let s= case (head a) of
           "t" -> Triangle; "c"-> Circle
  putStrLn (draw s)
```

- Haskell's Typisierung verdeckt dynamische Bindung
- Typ wird zur Übersetzungszeit berechnet.
  - Obwohl erst zur Laufzeit gebunden!
  - Implementierung von Klassen durch dictionaries

Existentielle Typen

- Idee aus der Logik: *Curry-Howard-Isomorphie*
- Getypter Lambda-Kalkül $\cong$ Intuitionistische Prädikatenlogik
- Beweis $\leftrightarrow$ Programm
- Allquantoren (höherer Ordnung) $\leftrightarrow$ Typvariablen
- Existenzquantoren $\leftrightarrow$

Existentielle Typen

- Idee aus der Logik: $Curry\text{-}Howard\text{-}Isomorphie$
- Getypter Lambda-Kalkül $\cong$ Intuitionistische Prädikatenlogik
- Beweis $\leftrightarrow$ Programm
- Allquantoren (höherer Ordnung) $\leftrightarrow$ Typvariablen
- Existenzquantoren $\leftrightarrow$ $ADTs!$

Abstract Data Types have Existential Type

- Polymorphie: allquantifizierte Typvariablen

```
forall a. data List a = Nil | Cons a (List a)"
```

- Typkonstruktor für alle Typen instanzierbar
- ADT: existenzquantifizierte Typvariablen

```
data T = forall a. Cons a (a-> Int)
```

- Typkonstruktor beschreibt einen unbestimmten Typ
- NB. nicht mehr Haskell98 (nur `ghc`, `hugs`).

- Beispiel: Emulation von dynamischer Bindung:

```
data T = forall a. Cons a (a-> Int)
ap (Cons x f) = f x
map ap [Cons [1,2,3] length,
        Cons 3 (5 +),
        Cons getLine (\_-> 0)]
```

- Heterogen Listen!
- Erweiterung: Klassenbeschränkung der Typvariablen:

```
data T = forall a. Show a=> Cons a (a-> Int)

instance Show T where
   show (Cons a _ ) = show a
```

Dynamische Bindung in Haskell?

- Beispiel: Klassen wie vorher, aber:

```
data ShapeT = forall s. Shape s=> Shape s
instance Shape ShapeT where
  draw (Shape s) = draw s

main :: IO ()
main = do
  a<- getArgs
  let s= case (head a) of
           "t" -> Shape Triangle; "c"-> Shape Circle

  putStrLn (draw s)
```

Dynamische Bindung in Haskell?

- Auflösung der Bindung zur Laufzeit
- Damit: heterogene Datenstrukturen
- Klassenhierarchie unnötig
- Keine echte Vererbung

Dynamische Bindung in Haskell?

- Auflösung der Bindung zur Laufzeit
- Damit: heterogene Datenstrukturen
- Klassenhierarchie unnötig
- Keine echte Vererbung
- Erweiterung nur a priori, nicht a posteriori
  - D.h. Datentypen müssen erweiterbar angelegt werden.

Polymorphie Höherer Ordnung

- Normale Polymorphie: allquantifizierte Typvariablen

```
data forall a. Maybe a = Nothing | Just a
fromJust :: forall a. Maybe a-> a
loop :: forall a. (a-> a)-> Int-> Int
```

- Allquantor immer außen.

Polymorphie Höherer Ordnung

- Normale Polymorphie: allquantifizierte Typvariablen

```
data forall a. Maybe a = Nothing | Just a
fromJust :: forall a. Maybe a-> a
loop :: forall a. (a-> a)-> Int-> Int
```

- Allquantor immer außen.
- Rang-n Polymorphie: Allquantor innen.

```
loop :: (forall a. a-> a)-> Int-> Int
```

- Rang 1: allquantifizierte Typvariablen
- Rang $n + 1$: Rang n auf der linken Seite eines Funktionstyps

Erstes Beispiel

- Zustandsübergangsmonaden, parametrisiert über Zustand `s`

```
type ST s a = s-> (a, s)
```

- Dazu: Zustandsbehaftete Berechnung ausführen

```
runST :: ST s a-> a
```

- Aber: Zustand sichtbar — `a` hängt von `s` ab.

```
let v = runST (newRef True) in runST (readVar v)
```

- Autsch — Deshalb:

```
runST :: forall a. (forall s. ST s a)-> a
```

- Zustand kann nicht entkommen, `a` von `s` unabhängig.

Generische Programmierung

- Generischer Fixpunktoperator:

```
forall f. (forall a. (a-> a)->
                     (f a-> f a))-> (Fix f-> Fix f)
```

- Definiert wie folgt:

```
data Fix f = Fix (f (Fix f))
```

- Mit dem Kind `(* -> *)-> *`
- Monomorphe Typen haben Kind `*`
- Polymorphe Typen haben Kind `*-> *`
- vgl. Konstruktorklassen

```
class Monad m where
  (>>=) :: m a-> (a-> m b)-> m b
```

Modellierung algebraischer Datentypen

- Listen als Rang-2 Datentyp:

```
type List a = (forall l. l-> (a-> l-> l)-> l)
```

  - `foldr` instantiiert `l`
  - Initialer Morphismus
- Führt zur Typentheorie — System F (Girard)
  - Polymorphie als Grundkonzept,
  - alg. Datentypen abgeleitet.

$$\mathbb{N} = \prod X.X \to (X \to X) \to X$$

Aufwand der Typüberprüfung

- Mit existentiellen Typen, Rang-n-Polymorphie:
  Typcheck wird unentscheidbar.
  - D.h. Typcheck kann divergieren!
- Aber: wen kümmert's?
- Auch Hindley-Milner (Haskell) ist exponentiell.
  - Kann divergieren.
  - Beispiel.

Zusammenfassung

- "Abstract types have existential type"
- (Limitierte) Modellierung von Objektorientierung
  - Fehlt: a posteriori Verfeinerung
- Rang-n Polymorphie
  - Beispiele: `runST`, generische Programmierung, ... $\rightsquigarrow$ Typentheorie
- Typüberprüfung wird unentscheidbar
  - ... aber schon Hindley-Milner-Typcheck ist exponentiell!

Fancy Types II: Abhängige Typen

In den nächsten Minuten:

- Was ist eigentlich Typentheorie?
- Typisierung in Haskell
- Verallgemeinerungen
- Abhängige Typen
- Beispiel: Epigram.

Typentheorie

- Was ist ein Typ?
- Programmiersprachen:

  $$\texttt{Term} : \texttt{Typ}$$

- Mathematik:
  - axiomatische Mengenlehre, universelle Algebra, etc. $\longrightarrow$ Syntax
  - $t : \tau$
- Klassifizierung von Ausdrücken

Typisierung in Haskell

- Regeln der Form $\dfrac{\Delta \vdash t : \tau}{\Gamma \vdash s : \sigma}$
- Für Haskell:
  - Variablen: $\dfrac{x : t}{x : t}$
  - Applikation: $\dfrac{A \vdash e : \sigma \to \tau \quad A \vdash e' : \sigma}{A \vdash ee' : \tau}$
  - Abstraktion: $\dfrac{A, x : \sigma \vdash e : \tau}{A \vdash \lambda x.e : \sigma \to \tau}$
- Rest ist syntaktischer Zucker.
- Aber Typunifikation fehlt!

Typsysteme

- Haskell: Einfach getypter λ-Kalkül mit Typvariablen
  - Funktionen höherer Ordnung
  - Typvariablen ganz außen quantifiziert
- Verallgemeinerung: warum nicht über alle Typen quantifizieren?
- Das große Problem: Russel's Paradoxon.
  - Es gibt keine Menge aller Mengen.
  - Es gibt auch keinen Typ aller Typen.

Eigenschaften von Typsystemen

- Church vs. Curry
  - Church: Implizit getypt, Typ mehrdeutig (e.g. Haskell)
  - Curry: Explizit getypt, Typ eindeutig (e.g. λ-Kalkül)
- Extensional vs. Intensional:
  - Extensional: $\forall x.fx = gx \Rightarrow f = g$
  - Intensional: $\forall x.fx = gx\neg \Rightarrow f = g$

Eigenschaften von Typsystemen

- Impredikativ vs. Predikativ:
  - Impredikativ: Quantisierung über allen Typen

    $$\mathbb{N} = \prod X.X \to (X \to X) \to X$$

    Unterscheidung zwischen Typen und Propositions
  - Predikativ: keine Quantisierung über alle Tyen

    Dafür: keinen Unterschied zwischen Typen und Aussagen.

    Es gibt einen Type : Type

Eigenschaften von Typsystemen

- Abhängige Typen vs. *stratified types*
  - Abhängige Typen: Typen können von Termen abhängen
  - *stratified:* Typen werden unabhängig von Termen aufgebaut.
- Typentheorien sind konstruktivistisch.
  - Satz vom ausgeschlossenen Dritten gilt nicht: $A \vee \neg A$

Verschiedene Typsysteme

- System F: impredikativ, zweiter Ordnung, intensional(?)
- Calculus of Constructions: impredikativ, höhere Ordnung, abhängige Typen
  - Implementiert in `Coq`
- Martin-Löf: predikativ, abhängige Typen, intensional
  - Implementiert in `Alfa`, `NuPRL` etc.
- Higher-Order Logic (Church): impredikativ, extensional
  - Implementiert in Isabelle

Abhängige Typen in Aktion: Epigram

- Epigram: prototypische Programmiersprache für abhängige Typen
- `http://www.dur.ac.uk/CARG/epigram.html`

Zusammenfassung

- Das große Problem: Russel-Paradox
- Verschiedene Typsysteme und ihre Eigenschaften
  - Weg um dieses Problem
- Abhängige Typen:
  - “Programming as Proving”
  - Typisierung nicht mehr entscheidbar

Zusammenfassung

Heute

- Zusammenfassung
- Ausblick
- Semesteraufgabe

Zusammenfassung I

- Monaden
  - Der `IO` Monade: `IORef`,
  - Zustandsübergangsmonaden (`ST`)
  - Kombination von Monaden, Monadentransformer
  - Fallbeispiel: monadische Parserkombinatoren

Zusammenfassung II

- Nebenläufige Programmierung in Haskell
  - lightweight threads auf Haskell-Ebene
  - Präemptives Multitasking im GHC
  - Synchronisation mit `MVar`
  - Darauf aufbauend: texttttChannel, Semaphoren, Monitore
  - Fallbeispiel: `talk`, Socketprogrammierung
- Fehlerbehandlung: `Exception`
  - Unscharfe Ausnahmen im `ghc`, asynchrone Ausnahmen
  - Dynamische Typen

Zusammenfassung III

- Sprachinteroperabilität
  - Haskell's Foreign Founction Interface
  - Marshalling Haskell-seitig
  - call-in und call-out
  - Werkzeuge: green card

Zusammenfassung III

- Sprachinteroperabilität
  - Haskell's Foreign Founction Interface
  - Marshalling Haskell-seitig
  - call-in und call-out
  - Werkzeuge: green card

- Effizienz
  - Endrekursion
  - Striktheit
  - Speicherlecks

Zusammenfassung IV

- GUIs und reaktive Programmierung
  - HTk vs. wxHaskell

Zusammenfassung IV

- GUIs und reaktive Programmierung
  - HTk vs. wxHaskell
- Domain-Specific Languages
  - Arrows

Zusammenfassung IV

- GUIs und reaktive Programmierung
  - HTk vs. wxHaskell
- Domain-Specific Languages
  - Arrows
- Fancy Types
  - Existentielle Typen
  - Polymorphie höherer Ordnung
  - Begriffe der Typentheorie
  - Abhängige Typen