Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 20.10.09: Einführung und Rückblick

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Organisatorisches

- Vorlesung: Di 8 – 10 MZH 7250
- Übung: Do 10 – 12 MZH 7210 (nach Bedarf)
- Scheinkriterien:
  - 5 Übungsblätter
  - Alle bearbeitet, insgesamt 40% (Notenspiegel PI3)
  - Übungsgruppen 2 – 4 Mitglieder
  - Ggf. Fachgespräch am Ende

Warum?

- Haskell: Nicht nur für Übungsaufgaben
- Funktionale Sprachen sind Innovationsinkubatoren
- Funktionale Sprachen behandeln Zukunftsthemen heute

Themen

- Monaden und fortgeschrittene Typen
  - Was ist eine "Monade"?
  - `IO` und andere Monaden
  - Konstruktorklassen, Rang-2-Polymorphie
- Fortgeschrittene Datenstrukturen
  - Der Zipper
  - Ströme, Graphen, unendliche Datenstrukturen
- Nebenläufigkeit
  - Leichtgewichte Threads in Haskell
  - Queues, Semaphoren, Monitore ...
  - State Transactional Memory
- The Future of Programming
  - Domänenspezifische Sprachen (DSLs)
  - Fancy Types
  - The Next Big Thing: F#, Scala

Ressourcen

- Haskell-Webseite: `http://www.haskell.org/`
- Büchereien:
  - Haskell 98 Libraries
  - Haskell Hierarchical Libraries
- Compiler:
  - Glasgow Haskell Compiler (`ghc`) (Version 6.10)
  - `http://www.haskell.org/ghc/`
- Bücher und Artikel
  - Siehe `http://www.informatik.uni-bremen.de/~cxl/lehre/asp.ws09/`
  - Ab morgen!

Rückblick 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 Haskell

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

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 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              getLine
           putStrLn s ⟷      >>= λs → putStrLn s
           echo               >> echo
```

- Oder auch:

```
echo = do { s← getLine; putStrLn s; echo }
```

- Nützlich:

```
(>>) :: IO t → IO u → IO u
f >> g ≡ f >>= λ_ → 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); reve
```

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 ++ "
```

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 | SeekFrom
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)
                  (λe→ putStr ("Error: " ++ (show
```

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)
                (λe→ 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 :: Permis
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_ (λf → do {mt← getModificationTime f;
                    if ((last f ≡ '~') &&
                        tdDay (diffClockTimes mt now)
1)
                    then removeFile f else return ()})
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

Fortgeschrittene Techniken der Funktionalen Programmierung

Vorlesung vom 27.10.09: Monads — The Inside Story

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Heute in diesem Theater

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

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
  - Einführung, Wiederholung
  - Zustände, Zustandsübergänge und `IO`
  - Reader/Writer, Nichtdeterminismus, Ausnahmen
  - Monadentransformer
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
- Teil IV: The Future of Programming

Zustandsübergangsmonaden

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

$$\mathsf{f} :: \mathsf{a} \to \mathsf{IO\ b} \quad \cong \quad \mathsf{f} :: (\mathsf{a}, \Sigma) \to (\mathsf{b}, \Sigma)$$

  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)
```

$$a \to ST\ s\ b = a \to s \to (b, s) \cong (a, s) \to (b, s)$$

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
- Nötig: 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

```
f >>= g = ST (uncurry (unwrap . g) . unwrap f)
return a = ST (λs → (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 eine Monade?

- `ST` modelliert imperative Konzepte.
- Beobachtung: Andere Konzepte können ähnlich modelliert werden:
  - 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

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

```
class Functor f where
  fmap :: (a → b) → (f a → f b)

instance Functor [ ] where -- Kein 'echtes' Haskell!
  fmap f [] = []
  fmap f (x:xs) = f x : map f xs
```

- Erweiterung des Typsystems (bleibt entscheidbar)
- Für Zustandstransformer ST:

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

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:

$$\begin{aligned} \text{return a} >\!\!>\!\!= \text{k} &= \text{k a} \\ \text{m} >\!\!>\!\!= \text{return} &= \text{m} \\ \text{m} >\!\!>\!\!= (\lambda \text{x} \rightarrow \text{k x} >\!\!>\!\!= \text{h}) &= (\text{m} >\!\!>\!\!= \text{k}) >\!\!>\!\!= \text{h} \end{aligned}$$

Beispiel: Speicher und Referenzen

- Signatur:

```
type Mem a
instance Mem Monad
```

- Referenzen sind abstrakt:

```
type Ref
newRef    :: Mem Ref
```

- Speicher liest/schreibt String:

```
readRef   :: Ref → Mem String
writeRef  :: Ref → String → Mem ()
```

Implementation der Referenzen

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

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

newRef = ST (λs→ (length s, s++[""]))
readRef r = ST (λs→ (s !! r, s))
writeRef r v = ST (λs→ ((),
                  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
```

- 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)
```

Flucht 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

- Signatur:

```
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

- IO-Aktionen, die nur
  - einmal durchgeführt werden sollen, und
  - von anderen IO-Aktionen unabhängig sind
  - Beispiel: Konfigurationsdatei lesen
- Alloziierung globaler Ressourcen (z.B. Referenzen).
- Debugging (traces, logfiles).
- Enjoy responsibly!

Benutzung von `unsafePerformIO`

- Alloziierung globaler Referenzen:

```
-- -- Generate a new identifier.
newId :: IO Int
newId = atomicModifyIORef r $ λi → (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
```

  - Schon vordefiniert (`Debug.Trace`).

Zusammenfassung & Ausblick

- Blick hinter die Kulissen von IO
- Monaden und andere Kuriositäten
- Referenzen
- unsafePerformIO
- Nächstes Mal: Mehr Monaden...

Fortgeschrittene Techniken der Funktionalen Programmierung

Vorlesung vom 03.11.09: Mehr über Monaden

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Heute gibt's:

- Monaden, mehr Monaden, und noch mehr Monaden
  - Die Monaden der Standardbücherei
  - Referenz: `http://www.haskell.org/all_about_monads/html`

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
  - Einführung, Wiederholung
  - Zustände, Zustandsübergänge und `IO`
  - Reader/Writer, Nichtdeterminismus, Ausnahmen
  - Monadentransformer
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
- Teil IV: The Future of Programming

Die Identitätsmonade

- Die allereinfachste Monade:

```
type Id a = a

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

Fehlermonaden

- Erste Nährung: Maybe
- Maybe kennt nur Nothing, daher strukturierte Fehler:

```
data Either a b = Left a  | Right b
type Error  a   = Either String a

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

- Nachteil: Fester Fehlertyp
- Lösung: Typklassen

Die Monade Control.Monad.Error

- Typklasse Error für Fehler

```
class Error a where
    noMsg  :: a
    strMsg :: String → a
```

- Fehlermonade parametrisiert über e:

```
class (Monad m) ⇒ MonadError e m where
    throwError :: e → m a
    catchError :: m a → (e → m a) → m a
```

```
instance MonadError (Either e) where
    throwError = Left
    (Left e) `catchError` handler = handler e
    a        `catchError` _       = a
```

Die Zustandsmonade

- Zustandsübergang als Funktion

```
newtype State s a = State {unwrap :: (s → (a,s))}

instance Monad (State s) where
    return a          = State $ λs → (a,s)
    (State g) >>= f = State (uncurry g . unwrap f)
```

- Nachteil 1: Zustandsübergang nicht-strikt (insbesondere lazy)!
  - Lösung: Strikter Zustandsübergang— Control.Monad.ST
- Nachteil 2: Zustandsübergang ≡ Funktion
  - Lösung: Typklassen

Die Monad Control.Monad.State

- Typklasse State für Zustand lesen/schreiben:

```
class MonadState m s where
    get :: m s
    put :: s → m ()
```

- Zustandsmonade parametrisiert über State:

```
instance MonadState (State s) s where
    get   = State $ λs → (s,s)
    put s = State $ λ_ → ((),s)
```

- Aber: manchmal liest man nur, manchmal schreibt man nur...

Die Lesemonade

- Intuition: Werte der Eingabe e lesen und verabeiten.
  - Lese-Teil der Zustandsmonade

```
newtype Reader e a = Reader (e → a)

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

- Eingabe wird nicht modifiziert.
- Beispiel: Lesen aus Symboltabelle (Gegenbeispiel: Datei)

Die Monade Control.Monad.Reader

- Wie vorher: Abstraktion der Leseoperationen
- Neu: Lokaler Zustand

```
class MonadReader e m where
    ask   :: m e
    local :: (e → e) → m a → m a

instance MonadReader (Reader e) where
    ask       = Reader id
    local f c = Reader $ λe → runReader c (f e)

asks :: (MonadReader e m) ⇒ (e → a) → m a
asks sel = ask ≫= return . sel
```

Die Schreibmonade

- Produziert einen Strom von Werten
- Kein Zugriff auf geschriebene Werte möglich
- Beispiel: “Logging”

```
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')
```

- Abstraktion: auch über Listen von Ausgabewerten

Die Monade Control.Monad.Writer

- Typklasse `Monoid`: Verallgemeinerte Listen

```
class (Monoid w, Monad m) ⇒ MonadWriter w m where
  pass   :: m (a, w → w) → m a
  listen :: m a → m (a, w)
  tell   :: w → m ()
```

```
instance MonadWriter (Writer w) where
  pass   (Writer ((a,f),w)) = Writer (a,f w)
  listen (Writer (a,w))     = Writer ((a,w),w)
  tell   s                  = Writer ((), s)
```

```
listens :: (MonadWriter w m) ⇒
           (w → w) → m a → m (a,w)
listens f m = do (a,w) ← m; return (a,f w)

censor :: (MonadWriter w m) ⇒
          (w → w) → m a → m a
censor f m = pass $ do a ← m; return (a,f)
```

Die Listenmonade

- Listen sind Monaden:

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

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

- Reihenfolge der Möglichkeiten relevant?

Der Monade Set

- Data.Set sind Monaden:

```
instance Monad Set where
  m >>= f   = Set.unions (Set.map f m)
  return x = Set.singleton x
  fail s    = Set.empty
```

- Nicht vordefiniert ...

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 $ λk → k a
  (Cont c) >>= f = Cont $
     λk → c (λa → runCont (f a) k)
```

Control.Monad.Cont

- callCC: GOTO für funktionale Sprachen

```
class (Monad m) ⇒ MonadCont m where
  callCC :: ((a → m b) → m a) → m a

instance MonadCont (Cont r) where
  callCC f = Cont $
    λk → runCont (f (λa → Cont $ λ_ → k a)) k
```

- Lieber nicht benutzen!

Exkurs: Was ist eigentliche eine Monade?

- Monade: Konstrukt aus Kategorientheorie
- Monade $\cong$ (verallgemeinerter) Monoid
- Monade: gegeben durch algebraische Theorien
  - Operationen endlicher (beschränkter) Aritität
  - Gleichungen
- Beispiele: Maybe, List, Set, State, ...
- Monaden in Haskell: computational monads
  - Strukturierte Notation für Berechnungsparadigmen
  - Beispiel: Rechner mit Fehler, Nichtdeterminismus, Zustand, ...

Kombination von Monaden: Das Problem

- Gegeben zwei Monaden:

```
class Monad m1 where ...

class Monad m2 where ...
```

- Es gelten weder

```
instance Monad (m1 (m2 a))
instance Monad (m2 (m1 a))
```

- Problem: Monadengesetze gelten nicht.
- Lösung: Nächste Vorlesung

Zusammenfassung

- Monaden sind praktische Abstraktion
- Wir haben kennengelernt:
  - Fehlermonaden: Maybe, Either, MonadError
  - Zustandsmonaden: State, ST, IO
  - Lese/Schreibmonade: ReaderMonad, WriterMonad
  - Nichtdeterminusmus: [a], Data.Set
  - Explizite Sprünge: Continuation
- Wichtiges Strukturierungsmittel für funktionale Programme
- Kombination bereitet (noch) Probleme ...

Fortgeschrittene Techniken der Funktionalen Programmierung

Vorlesung vom 10.11.09: Monadentransformer

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Heute gibt's:

- Eine Monade ist gut, mehrere Monaden sind besser
- Kombination von Monaden
- Monadentransformer

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
  - Einführung, Wiederholung
  - Zustände, Zustandsübergänge und `IO`
  - Reader/Writer, Nichtdeterminismus, Ausnahmen
  - Monadentransformer
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
- Teil IV: The Future of Programming

Kombination von Monaden: Das Problem

- Gegeben zwei Monaden:

```
class Monad m1 where ...

class Monad m2 where ...
```

- Es gelten weder

```
instance Monad (m1 (m2 a))
instance Monad (m2 (m1 a))
```

- Problem: Monadengesetze gelten nicht.

Monadentransformer

- Monadentransformer:
  - Erweiterbare Monade
  - Monade mit Loch

```
newtype M a = ...
instance Monad M where
  f >>= g  = ...
  return x = ...
```

$\Rightarrow$

```
newtype M m a = ...
instance Monad m => Monad M m where
  f >>= g  = ...
  return x = ...
```

Monadentransformer: Beispiele

1. Beispiel: Zustandsmonadentransformer

```
type StateT s m a = s → m (s, a)
```

   - Zustandsbasierte Berechnungen in einer anderen Monade m
   - StateT s Identity ist Zustandsmonade

2. Beispiel: Fehlermonadentransformer

```
type ErrorT e m a = m (Either e a)
```

   - Fehlerbehaftete Berechnungen in einer anderen Monade m
   - ErrorT e Identity ist Fehlermonade

Reihenfolge beachten!

- Kombination von `State` und `Error`
- Erst Zustand, dann Fehler:

```
type ErrorState s a =
  ErrorT (StateT s Id) = Either String (s→ (s ,a))
```

  - Berechung m :: ErrorState s a: Fehler oder zustandsbehaftet
  - E.g. Fehler in Haskell

- Erst Fehler, dann Zustand:

```
type StateError s a = s→ (s, Either String a)
```

  - Berechung m :: StateError s a: Immer zustandsbehaftet, Resultat Fehler oder normal
  - E.g. Fehler in imperativen Sprachen

Standardtransformer

Standard-Monade	Transformer	Standard-Typ	Transformierter Typ
Error	ErrorT	Either e a	m (Either e a)
State	StateT	s → (a,s)	s → m (a,s)
Reader	ReaderT	r → a	r → m a
Writer	WriterT	(a,w)	m (a,w)
Cont	ContT	(a → r) → r	(a → m r) → m r

Quelle: `http://www.haskell.org/all_about_monads/`

Fallbeispiel: ein Modularer Interpreter

- Ziel: Interpreter für eine einfache imperative Sprache
- Modularer Aufbau:
  1. Nur Zustand
  2. Ausgabe
  3. Eingabe
  4. Fehler und Fehlerbehandlung

Zusammenfassung

- Warum Monaden kombinieren?
  - Typ definiert Effekt
  - E.g. WriterT — nur Logging, kein Zustand
  - Pragmatisch: die IO-Monade (imperativ)
- Vorteile Monadentransfomer:
  - Erlauben modulare Kombination von Monaden
  - Standard-Bücherei (Monad Template Library) bietet Standard-Monaden als praktischen Bausatz
- Nachteile Monadentransformer:
  - Für neue Transformer Kombination mit allen anderen zu bedenken!
  - Nicht kompositional
- Nächste Woche: ?

Fortgeschrittene Techniken der Funktionalen Programmierung

Vorlesung vom 17.11.09: Der Zipper

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
- Teil II: Fortgeschrittene Datenstrukturen
  - Der Zipper
  - Ströme, Graphen, unendliche Datenstrukturen
- Teil III: Nebenläufigkeit
- Teil IV: The Future of Programming

Das Problem

- Funktional = kein Zustand
- Wie destruktiver Update?
  - Manipulation innerhalb einer Datenstruktur
  - Beispiel: ZeilenorientierterEditor, Abstrakte Syntax
- Lösung: Der Zipper
  - Keine feste Datenstruktur, sondern ein Schema

Ein einfacher Editor

- Datenstrukturen:

```
type Text   = [String]
data Pos    = Pos { line :: Int, col :: Int}
data Editor = Ed  { text   :: Text
                  , cursor :: Pos }
```

- Operationen: Cursor bewegen (links)

```
go_left :: Editor → Editor
go_left Ed{text= t, cursor= c}
  | col c ≡ 0 = error "At start of line"
  | otherwise =
      Ed{text= t, cursor=c{col= col c- 1}}
```

Beispieloperationen

- Text rechts einfügen:

```
insert_right :: Editor → String → Editor
insert_right Ed{text= t, cursor= c} text =
  let (as, bs) = splitAt (col c) (t !! line c)
  in  Ed{text= updateAt (line c) t
                        (as ++ text++ bs),
         cursor= c}
```

```
updateAt :: Int → [a]→ a→ [a]
updateAt n as a = case splitAt n as of
  (bs, [])     → error "updateAt: list too short."
  (bs, _:cs) → bs ++ a : cs
```

- Problem: Aufwand für Manipulation

Manipulation strukturierter Datentypen

- Anderes Beispiel: *n*-äre Bäume (rose trees)

```
data Tree a = Leaf a
            | Node [Tree a]
            deriving Show
```

  - Bsp: Abstrakte Syntax von einfachen Ausdrücken

- Update auf Beispielterm $t = a * b - c * d$: ersetze b durch $x + y$

```
t = Node [ Leaf "-"
         , Node [Leaf "*", Leaf "a", Leaf "b"]
         , Node [Leaf "*", Leaf "c", Leaf "d"]
         ]
```

Der Zipper

- Idee: Kontext nicht wegwerfen!
- Nicht: **type** Path = [Int]
- Sondern:

```
data Ctxt a = Empty
            | Cons [Tree a] (Ctxt a) [Tree a]
```

  - Kontext ist 'inverse Umgebung' (*"Like a glove turned inside out"*)

- Loc a ist Baum mit Fokus

```
newtype Loc a = Loc (Tree a, Ctxt a)
```

  - Warum newtype?

Zipping Trees: Navigation

- Fokus nach links

```
go_left :: Loc a → Loc a
go_left (Loc(t, c)) = case c of
 Empty → error "go_left at empty"
 Cons (l:le) up ri → Loc(l, Cons le up (t:ri))
 Cons [] _ _       → error "go_left of first"
```

- Fokus nach rechts

```
go_right :: Loc a → Loc a
go_right (Loc(t, c)) = case c of
 Empty → error "go_right at empty"
 Cons le up (r:ri) → Loc(r, Cons (t:le) up ri)
 Cons _ _ []       → error "go_right of last"
```

Zipping Trees: Navigation

- Fokus nach oben

```
go_up :: Loc a → Loc a
go_up (Loc (t, c)) = case c of
  Empty → error "go_up of empty"
  Cons le up ri →
    Loc (Node (reverse le ++ t:ri), up)
```

- Fokus nach unten

```
go_down :: Loc a → Loc a
go_down (Loc (t, c)) = case t of
  Leaf _ → error "go_down at leaf"
  Node [] → error "go_down at empty"
  Node (t:ts) → Loc (t, Cons [] c ts)
```

Zipping Trees: Navigation

- Hilfsfunktion:

```
top :: Tree a→ Loc a
top t = (Loc (t, Empty))
```

- Damit andere Navigationsfunktionen:

```
path :: Loc a→ [Int]→ Loc a
path l [] = l
path l (i:ps)
  | i ≡ 0 = path (go_down l) ps
  | i > 0 = path (go_left l) (i−1) ps
```

Einfügen

- Einfügen: Wo?
- Links des Fokus einfügen

```
insert_left :: Tree a → Loc a → Loc a
insert_left t1 (Loc (t, c)) = case c of
  Empty → error "insert_left: insert at empty"
  Cons le up ri → Loc(t, Cons (t1:le) up ri)
```

- Rechts des Fokus einfügen

```
insert_right :: Tree a → Loc a → Loc a
insert_right t1 (Loc (t, c)) = case c of
  Empty → error "insert_right: insert at empty"
  Cons le up ri → Loc(t, Cons le up (t1:ri))
```

- Unterhalb des Fokus einfügen

```
insert_down :: Tree a → Loc a → Loc a
insert_down t1 (Loc(t, c)) = case t of
  Leaf _ → error "insert_down: insert at leaf"
  Node ts → Loc(t1, Cons [] c ts)
```

Ersetzen und Löschen

- Unterbaum im Fokus ersetzen:

```
update :: Tree a → Loc a → Loc a
update t (Loc (_, c)) = Loc (t, c)
```

- Unterbaum im Fokus löschen: wo ist der neue Fokus?
  1. Rechter Baum, wenn vorhanden
  2. Linker Baum, wenn vorhanden
  3. Elternknoten

```
delete :: Loc a → Loc a
delete (Loc(_, p)) = case p of
  Empty → Loc(Node [], Empty)
  Cons le up (r:ri) → Loc(r, Cons le up ri)
  Cons (l:le) up [] → Loc(l, Cons le up [])
  Cons [] up [] → Loc(Node [], up)
```

  - *"We note that delete is not such a simple operation."*

Schnelligkeit

- Wie schnell sind Operationen?

Schnelligkeit

- Wie schnell sind Operationen?
  - Aufwand: go_left $O(left(n))$, alle anderen $O(1)$.
- Warum sind Operationen so schnell?

Schnelligkeit

- Wie schnell sind Operationen?
  - Aufwand: go_left $O(\mathit{left}(n))$, alle anderen $O(1)$.
- Warum sind Operationen so schnell?
  - Kontext bleibt erhalten
  - Manipulation: reine Zeiger-Manipulation

Der Zipper als Monade

- LocM ist Zustandsmonade mit Zustand Loc

```
newtype LocM a b = LocM {locSt :: State (Loc a) b}
  deriving (Functor, Monad, MonadState (Loc a))
```

- Startfunktion

```
run :: LocM a b→ Tree a→ (b, Loc a)
run l t = runState (locSt l) (top t)
```

- Zugriff auf Baum im Fokus

```
current :: LocM a (Tree a)
current = do Loc(l, c)← get; return l
```

- Navigation

```
left :: LocM a ()
left = modify go_left
-- etc.
```

Zipper Monad: Manipulation

- Löschen:

```
del :: LocM a ()
del = modify delete
```

- Update:

```
upd :: Tree a → LocM a ()
upd t = modify (update t)
```

- Einfügen (bsp rechts):

```
insr :: Tree a → LocM a ()
insr t = modify (insert_right t)
```

- Hilfsprädikat: right möglich?

```
has_right :: Loc a → Bool
has_right (Loc(l, c)) = case c of
  (Cons (_:_) _ _) → True; _ → False
```

```
has_ri :: LocM a Bool
has_ri = gets has_right
```

Zipper für andere Datenstrukturen

- Binäre Bäume:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

- Kontext:

```
data Ctxt a = Empty
            | Le (Ctxt a) (Tree a)
            | Ri (Tree a) (Ctxt a)
```

```
newtype Loc a = Loc (Tree a, Ctxt a)
```

Tree-Zipper: Navigation

- Fokus nach links

```
go_left :: Loc a → Loc a
go_left (Loc(t, ctx)) = case ctx of
 Empty  → error "go_left at empty"
 Le c r → error "go_left of left"
 Ri l c → Loc(l, Le c t)
```

- Fokus nach rechts

```
go_right :: Loc a → Loc a
go_right (Loc(t, ctx)) = case ctx of
 Empty  → error "go_right at empty"
 Le c r → Loc(r, Ri t c)
 Ri _ _ → error "go_right of right"
```

Tree-Zipper: Navigation

- Fokus nach oben

```
go_up :: Loc a → Loc a
go_up (Loc(t, ctx)) = case ctx of
  Empty  → error "go_up of empty"
  Le c r → Loc(Node t r, c)
  Ri l c → Loc(Node l t, c)
```

- Fokus nach unten links

```
go_down_left :: Loc a → Loc a
go_down_left (Loc(t, c)) = case t of
  Leaf _   → error "go_down at leaf"
  Node l r → Loc(l, Le c r)
```

- Fokus nach unten rechts

```
go_down_right :: Loc a → Loc a
go_down_right (Loc(t, c)) = case t of
  Leaf _   → error "go_down at leaf"
  Node l r → Loc(r, Ri l c)
```

Tree-Zipper: Einfügen und Löschen

- Einfügen links

```
ins_left :: Tree a → Loc a → Loc a
ins_left t1 (Loc(t, ctx)) = Loc(t, Ri t1 ctx)
```

- Einfügen rechts

```
ins_right :: Tree a → Loc a → Loc a
ins_right t1 (Loc(t, ctx)) = Loc(t, Le ctx t1)
```

- Löschen

```
delete :: Loc a → Loc a
delete (Loc(_, c)) = case c of
  Empty  → error "delete of empty"
  Le c r → Loc(r, c)
  Ri l c → Loc(l, c)
```

  - Neuer Fokus: anderer Teilbaum

Tree-Zipper: Variation

- Binäre Bäume, Werte im Knoten:

```
data Tree a = Nil | Node (Tree a) a (Tree a)
```

- Kontext enthält Knotenwert

```
data Ctxt a  = Empty
             | Le (Ctxt a) a (Tree a)
             | Ri (Tree a) a (Ctxt a)
```

- Funktionen ähnlich, aber:
  - delete total, löscht Knoteninhalt im Kontext

Zipping Lists

- Listen:

```
data List a = Nil | Cons a (List a)
```

- Damit:

```
data Ctxt a = Empty | Snoc (Ctxt a) a
```

- Listen sind ihr ‘eigener Kontext’ :

$$\texttt{List a} \cong \texttt{Ctxt a}$$

Zipping Lists: Fast Reverse

- Listenumkehr schnell:

```
fastrev :: [a] → [a]
fastrev xs = rev xs []

rev :: [a] → [a] → [a]
rev []     as = as
rev (x:xs) as = rev xs (x:as)
```

- Zweites Argument von rev: Kontext
  - Liste der Elemente davor in umgekehrter Reihenfolge

Zusammenfassung

- Der Zipper
  - Manipulation von Datenstrukturen
  - Zipper = Kontext + Fokus
  - Effiziente destruktive Manipulation
- Nachteile
  - Nicht richtig generisch — Schema, keine Bücherei
  - Viel schematischer Code für jeden Datentyp
  - Abhilfe: Generic Zipper
- Nächstes Mal: Graphen, Ströme, unendliche Datenstrukturen

Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 24.11.09: Unendliche Datentypen und Graphen

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
- Teil II: Fortgeschrittene Datenstrukturen
  - Der Zipper
  - Ströme, Graphen, unendliche Datenstrukturen
- Teil III: Nebenläufigkeit
- Teil IV: The Future of Programming

Das Tagesmenü

- Reprise: Ströme (uendliche Listen)
- Doppelt verkettete Listen
- Graphen

Unendliche Datenstrukturen: Ströme

- Ströme: Unendliche Listen

```
data Stream a = Stream { hd :: a
                       , tl :: Stream a
                       }
```

- Observatoren:

```
hd :: Stream a → a
tl :: Stream a → Stream a
```

Bsp: Fibonacci-Zahlen

- Fibonacci-Zahlen als Strom
- Sei `fibs :: [Integer]` Strom aller Fib'zahlen:

```
            fibs  1  1  2  3  5  8 13 21 34 55
       tail fibs  1  2  3  5  8 13 21 34 55
tail (tail fibs)  2  3  5  8 13 21 34 55
```

- Damit ergibt sich:

```
fibs :: Stream Integer
fibs = Stream 1 (Stream 1 $ zipS (+) fibs (tl fibs))
```

- `n`-te Fibonaccizahl mit `ith n fibs`
- Aufwand: linear, da `fibs` nur einmal ausgewertet wird.

Doppelt Verkettete Listen

- In Haskell wie in Java/C: Zeiger auf Vorgänger, Nachfolger

```
data DList a = DLNode { prev :: DList a
                      , val  :: a
                      , next :: DList a }
             | DLEmpty
```

  - **deriving** (Eq, Show) ???

- Kein freier Datentyp: es gelten Invarianten

```
d /= DLEmpty && next d /= DLEmpty
                       ⇒ prev (next d)= d
d /= DLEmpty && prev d /= DLEmpty
                       ⇒ next (prev d)= d
```

Listen erzeugen

- Doppelt verkettete Listen erzeugen:

```
fromList :: [a] → DList a
fromList as = mkDList DLEmpty as

mkDList :: DList a → [a] → DList a
mkDList currprev []     = DLEmpty
mkDList currprev (x:xs) =
  let here = DLNode currprev x (mkDList here xs)
  in  here
```

- Problem: Knoten einfügen/löschen
  - Einfache falsche Lösung
  - Richtige Lösung ist $O(n)$

Zusammenfassung

- Vorteile:
  - Vorgänger/Nachfolger $O(1)$
- Nachteile:
  - Einfügen/Löschen $O(n)$

Graphen als unendliche Datentypen

- Ein Graph ist eine Liste von Knoten

```
data Node a b   = Node a [Vertex a b]
data Vertex a b = Vertex b (Node a b)
type Graph a b  = [Node a b]
```

- Damit Beispielgraph:

```
g :: Graph String String
g = let n1 = Node "a" [Vertex "R" n2]
        n2 = Node "b" [Vertex "L" n1, Vertex "D" n3]
        n3 = Node "c" [Vertex "U" n1]
    in g
```

- Problem: Einfügen/Löschen (King & Launchbury, 1995)

Graphen als induktive Datenstrukturen

- Martin Erwig (2001)— Functional Graph Library (FGL)
- Ideen:
  1. Knoten haben explizite Identität
  2. Graph ist induktiv definiert
- Ein Graph ist
  - entweder leer
  - oder Erweiterung eines Graphen (Kontext)

```
type Node     = Int
type Adj b    = [(b, Node)]
type Ctx a b = (Adj b, Node, a, Adj b)

data Gr a b   = Empty | Ctx a b :& Gr a b
```

Pattern Matching

- Graph kein freier Datentyp — Beispiel
- Datentyp Graph muss abstrakt sein
- Fallunterscheidung auf Konstruktoren von Graph nicht möglich
- Fälle:
  - Leerer Graph

```
isEmpty :: Gr a b → Bool
```

  - Nicht-leerer Graph: Kontext plus Rest

```
matchAny :: Gr a b → (Ctx a b, Gr a b)
```

Einfache Funktionen

- Match auf einen bestimmten Knoten:

```
match :: Node → Gr a b → (Maybe (Ctx a b), Gr a b)
```

Invariante:

$$match\, v\, g = (Just(is, w, l, os), h) \Longrightarrow v = w$$

- Map:

```
gmap :: (Ctx a b → Ctx c d) → Gr a b → Gr c d
gmap f g | isEmpty g = Empty
         | otherwise = f c :& (gmap f g') where
             (c, g') = matchAny g
```

- Damit Umkehr aller Kanten:

```
swap :: Ctx a b → Ctx a b
swap (p, v, l, s) = (s, v, l, p)
grev :: Gr a b → Gr a b
grev = gmap swap
```

Beweise von Eigenschaften

- Datentyp Gr a b ist nicht frei, aber induktiv
- Induktion als zulässiges Beweisprinzip:

$$\frac{\mathit{empty}\, g \longrightarrow P\, g \quad \forall c\, g.\, P\, g \longrightarrow P(c : \&g)}{\forall g.P\, g}$$

- Damit zeigen:

$$\begin{array}{rcll} \mathit{gmap}\, f\, .\mathit{gmap}\, f' & = & \mathit{gmap}\, (f\, .f') & \text{(gmap fusion)} \\ \mathit{grev}\, .\mathit{grev} & = & \mathit{id} & \text{(grev inv)} \end{array}$$

Tiefensuche

- Aufspannenden Baum in Tiefensuche

```
data Tree a = Tree a [Tree a]

df :: [Node]→ Gr a b→ ([Tree Node], Gr a b)
df [] g       = ([], g)
df (v:vs) g = case match v g of
     (Just c, h) → (Tree v t1: t2, g2)
                 where (t1, g1)= df (suc c) h
                       (t2, g2)= df vs g1
     (Nothing, h) → df vs h

dff :: [Node]→ Gr a b→ [Tree Node]
dff vs g = fst (df vs g)
```

- Anwendung: SCC (stark verbundene Komponenten)

Breitensuche

- Aufspannenden Baum in Breitensuche
  - Problem: Baum wächst nach ‘unten’ — daher Pfade

```
type Path  = [Node]
type RTree = [Path]

bft :: Node→ Gr a b→ RTree
bft v = bf [[v]]

bf :: [Path]→ Gr a b→ RTree
bf [] g = []
bf (p@(v:_):ps) g = case match v g of
  (Just c, h)  → p:bf (ps ++ map (:p) (suc c)) h
  (Nothing, h) → bf ps h
```

- Verbesserung: Queue Path statt [Path] benutzen
- Anwendung: kürzester Pfad

FGL als Bücherei

- Viele weitere Algorithmen
- Graphen als Klasse, verschiedene Implementationen
- Hackage: Data.Graph.Inductive
- Mehr hier:
  `http://web.engr.oregonstate.edu/~erwig/fgl/haskell/`

Zusammenfassung

- Unendliche Datenstrukturen realisiert durch Referenzen
- Programmierung: Observatoren/Destruktoren vs. Konstruktoren
- Beispiel: doppelt verkettete Listen
- Graphen in Haskell
  - Beispiel für induktive, aber nicht freie Datenstruktur
  - Kompakte Darstellung, effiziente Algorithmen möglich
- Nächste Woche: Nebenläufigkeit

Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 01.12.09: Grundlagen der Nebenläufigkeit in Haskell

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
  - Grundlagen
  - Abstraktionen und Ausnahmebehandlung
  - Software Transactional Memory
- Teil IV: The Future of Programming

Heute gibt's hier

Nebenläufigkeit

- Grundkonzepte
- Implementation in Haskell
- Basiskonzepte

Konzepte der Nebenläufigkeit

Thread (lightweight process)	vs.	Prozess
Programmiersprache/Betriebssystem		Betriebssystem
(z.B. Java, Haskell, Linux)		
gemeinsamer Speicher		getrennter Speicher
Erzeugung billig		Erzeugung teuer
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`

Threads in Haskell: Concurrent Haskell

- Sequentielles Haskell: Reduktion eines Ausdrucks
  - Compiler legt Reihenfolge fes (outermost leftmost — verzögerte Auswertung)
- Nebenläufiges Haskell: Reduktion eines Ausdrucks an mehreren Stellen
- `ghc` und `hugs` implementieren Haskell-Threads
- `ghc`: präemptiv, `hugs`: kooperativ
- Modul `Control.Concurrent` enthält Basisfunktionen
- Wenige Basisprimitive, darauf aufbauend Abstraktionen

Wesentliche Typen und Funktionen

- Jeder Thread hat einen Identifier: abstrakter Typ `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 ()`

Rahmenbedingungen

- Zeitscheiben:
  - Tick: Default $20ms$
  - Contextswitch pro Tick bei Heapallokation
  - Änderungen per Kommandozeilenoptionen: `+RTS -V¡time¿ -C¡time¿`
- Blockierung:
  - Systemaufrufe blockieren alle Threads
    - Mit threaded library (`-threaded`) nicht alle
  - Aber: Haskell Standard-IO blockiert nur den aufrufenden Thread

Concurrent Haskell — erste Schritte

- Ein einfaches Beispiel:

```
write :: Char → IO ()
write c = putChar c ≫ write c

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

- Ausgabe `ghc`: $(\mathtt{X}^*|\mathtt{O}^*)^*$
- Ausgabe `hugs`: $(\mathtt{X}^*|\mathtt{O}^*)$

Synchronisation mit MVars

- Basissynchronisationmechanismus in Concurrent Haskell
  - Alles andere abgeleitet
- MVar a veränderbare Variable (vgl. IORef a)
- Entweder leer oder gefüllt mit Wert vom Typ a
- Verhalten beim Lesen und Schreiben

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

  - NB. Aufwecken blockierter Prozesse einzeln in FIFO

Basisfunktionen MVars

- 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 ()
```

Abgeleitete Funktionen MVars

- Nicht-blockierendes Lesen/Schreiben:

```
tryTakeMVar :: MVar a → IO (Maybe a)
tryPutMVar  :: MVar a→ a→ IO Bool
```

- Änderung der MVar:

```
swapMVar    :: MVar a → a → IO a
withMVar    :: MVar a → (a → IO b) → IO b
modifyMVar  :: MVar a → (a → IO (a, b)) → IO b
```

  - Achtung: race conditions

Ein einfaches Beispiel ohne Synchronisation

- Nebenläufige Eingabe von der Tastatur

```
import Control.Monad(forever, replicateM_)
import Control.Concurrent

echo :: String → IO ()
echo p = forever $ do
  putStrLn ("*** Please enter line for "++p)
  line ← getLine
  n ← randomRIO (1,100)
  replicateM_ n $ putStr (p++ ":"++ line++" ")

main :: IO ()
main = forkIO (echo "2") ≫ echo "1"
```

- Problem: gleichzeitige Eingabe

Ein einfaches Beispiel ohne Synchronisation

- Nebenläufige Eingabe von der Tastatur

```
import Control.Monad(forever, replicateM_)
import Control.Concurrent

echo :: String → IO ()
echo p = forever $ do
  putStrLn ("*** Please enter line for "++p)
  line ← getLine
  n ← randomRIO (1,100)
  replicateM_ n $ putStr (p++ ":"++ line++" ")

main :: IO ()
main = forkIO (echo "2") ≫ echo "1"
```

- Problem: gleichzeitige Eingabe
- 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 = forever $ do
  takeMVar flag
  putStrLn ("*** Please enter line " ++ p)
  line ← getLine
  n ← randomRIO (1,100)
  replicateM_ n $ putStr (p ++ ":" ++ line ++ " ")
  putMVar flag ()

main :: IO ()
main = do flag ← newMVar ()
          forkIO (echo flag "3") ≫ forkIO (echo flag
echo flag "1"
```

Das Standardbeispiel

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

Speisende Philosophen

```
philo :: [MVar ()] → Int → IO ()
philo chopsticks i = forever $ do
    let num_phil = length (chopsticks)
    -- Thinking:
    putStrLn ("Phil #" ++ show i ++ " thinks...")
    randomRIO (10, 200) >>= threadDelay
    -- Get ready to eat:
    takeMVar (chopsticks !! i)
    takeMVar (chopsticks !! ((i+1) `mod` num_phil))
    -- Eat:
    putStrLn ("Phil #" ++ show i ++ " eats...")
    randomRIO (10, 200) >>= threadDelay
    -- Done eating:
    putMVar (chopsticks !! i) ()
    putMVar (chopsticks !! ((i+1) `mod` num_phil)) ()
```

Speisende Philosophen

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

```
main = do
  a:_ ← getArgs
  let num= read a
  chopsticks ← replicateM num $ newMVar ()
  mapM_ (forkIO . (philo chopsticks)) [0.. num-1]
  block
```

- Hilfsfunktion `block`: blockiert aufrufenden Thread

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

- NB: Hauptthread terminiert — Programm terminiert!

Abstraktion: 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

- Verlassen des kritischen Abschnitts
- 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 ()
```

Zusammenfassung

- Concurrent Haskell bietet
  - Threads auf Quellsprachenebene
  - Synchronisierung mit `MVars`
  - Durch schlankes Design einfache Implementierung
- Funktionales Paradigma erlaubt Abstraktionen
  - Beispiel: Semaphoren
- Nächste Woche: Kanäle und Ausnahmen.

Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 08.12.09:
Nebenläufigkeit in Haskell: Abstraktionen und Ausnahmen

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
  - Grundlagen
  - Abstraktionen und Ausnahmebehandlung
  - Software Transactional Memory
- Teil IV: The Future of Programming

Tagesmenü

- Abstraktionen: Kanäle
- Fallbeispiel:
  - Talk
- Ausnahmebehandlung:
  - Erweiterbare Ausnahmen
  - Unscharfe Ausnahmen
  - Asynchrone Ausnahmen

Kanäle

- Typsicheres Lesen/Schreiben in FIFO-Ordnung
- Blockiert wenn leer

```
data Chan a ...
newChan   :: IO (Chan a)
writeChan :: Chan a → a → IO ()
readChan  :: Chan a → IO a
```

  - Bonus: Duplizierbar (“Broadcast”)

Kanäle

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

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

  - Hier MVar, um Lesen/Schreiben zu synchronisieren

- 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)
```

In einen Kanal schreiben

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

```
writeChan :: Chan a → a → IO ()
writeChan (Chan _ write) val = do
  new_hole ← newEmptyMVar
  old_hole ← takeMVar write
  putMVar old_hole (ChItem val new_hole)
  putMVar write new_hole
```

- Kann nicht blockieren — write immer gefüllt.
- Original-Code benutzt modifyMVar — Ausnahmesicher!

Aus Kanal lesen

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

```
readChan :: Chan a → IO a
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

Neuen Kanal erzeugen

- Lese-Ende = Schreib-Ende

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

Weitere Kanalfunktionen

- Zeichen wieder vorne einhängen:

```
unGetChan :: Chan a → a → IO ()
```

- Kanal duplizieren (Broadcast):

```
dupChan :: Chan a → IO (Chan a)
```

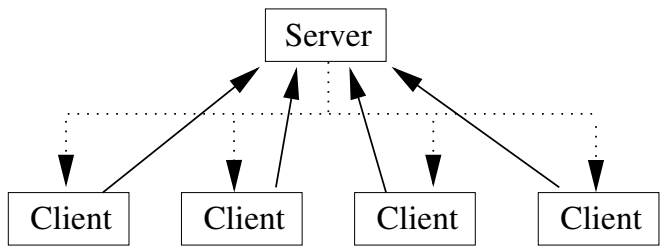
- Kanalinhalt als (unendliche) Liste:

```
getChanContents :: Chan a → IO [a]
```

- Auswertung terminiert nicht, sondern blockiert

Fallbeispiel: 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`

- Sockets:

```
type Socket
data PortID = Service String -- z.B. "ftp"
            | PortNumber PortNumber
            | UnixSocket String -- Socket mit Namen
type Hostname = String
instance Num PortNumber
```

- Zustandsbasiert:

```
listenOn   :: PortID -> IO Socket
accept     :: Socket -> IO (Handle, Hostname, PortNum
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
- Client: `telnet`

Talk 0.1: Hauptprogramm

```
main = do
  a:_ ← getArgs
  let p = fromInteger (read a)
  s ← listenOn (PortNumber p)
  ch ← newChan
  loop s ch
```

Talk 0.1: Hauptschleife

```
loop s ch = forever $ do
  (handle, wh, p) ← accept s
  hSetBuffering handle NoBuffering
  putStrLn $ "New connection from " ++ wh ++
             " on port " ++ show p
  ch2    ← dupChan ch
  forkIO (newUser handle ch2)
```

Talk 0.1: Benutzerprozess

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

Talk 0.1: Zusammenfassung

Nachteile:

- Nachrichten stauen sich im Kanal
- Keine Fehlerbehandlung
- Benutzer anonym

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: Hauptschleife

```
loop s ch = forever $ 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)
                (λ_ → hClose handle))
```

- Fehlerbehandlung für newUser (kein guter Stil)
- SIGPIPE ignorieren

Talk 0.2: Benutzerprozess

Teil 1: Anmeldeprozedur

```
newUser s wh msgch = do
  hPutStrLn s "Hello there. Please send your nickname
  nick <- do nm <- hGetLine s
             return (filter (not . isControl) nm)
  hPutStrLn s $ "Nice to meet you, "++ nick++ "!"
  writeChan msgch $ nick++ "@"++ wh++ " has joined."
```

(Fortsetzung)

Talk 0.2: Benutzerprozess

Teil 2: Hauptschleife:

```
wp ← forkIO write
catch (read ((nick ++ ": ") ++)) $ λe→ do
  killThread wp
  writeChan msgch $
    if isEOFError e then nick++ "@"++ wh++ " has le
    else nick++ "@"++ wh++ " left hastily ("++
                    ioeGetErrorString e++ ")"
  hClose s where
read :: (String→ String)→ IO ()
read f = forever $
              hGetLine s >>= writeChan msgch. f
write :: IO ()
write = forever $ readChan msgch >>= hPutStrLn s
```

Talk 0.2: Zusammenfassung

Vorteile:

- Robust
- Fehlerbehandlung für Benutzerprozess
- Anmeldeprozedur: Benutzer hat Namen

Talk 0.2: Zusammenfassung

Vorteile:

- Robust
- 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
- Aber: Auswertungsreihenfolge in Haskell98 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.

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)
- Um Erweiterbarkeit zu gewährleisten:
  - Typklasse Exception, alle Ausnahmen sind Instanzen
  - Achtung, erst seit ghc 6.10.
- Achtung: per default normale Ausnahmen (Haskell98) definiert
  - Überlagerung durch Import oder Disambiguierung
- Ausnahmen fangen:

```
catch :: Exception e⇒ IO a→ (e → IO a)→ IO a
try :: Exception e⇒ IO a → IO (Either e a)
```

Vorsicht bei Ausnahmen

- 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)
      (λe → ...)
```

- 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

- Kanäle: Nützliche Kommunikationsabstraktion
- Unscharfe Ausnahmen:
  - Können in beliebigem Code auftreten
  - Werden im IO-Monaden gefangen
- Asynchrone Ausnahmen:
  - Werden in anderem Thread ausgelöst
- Ausnahmebehandlung:
  - Essentiell für robuste Programmierung
  - Nur Ausnahmen fangen, die man behandelt!
  - Fehlerabfrage ersetzt keine Ausnahmebehandlung

Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 15.12.09:
Nebenläufigkeit in Haskell: Software Transactional Memory

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
  - Grundlagen
  - Abstraktionen und Ausnahmebehandlung
  - Software Transactional Memory
- Teil IV: The Future of Programming

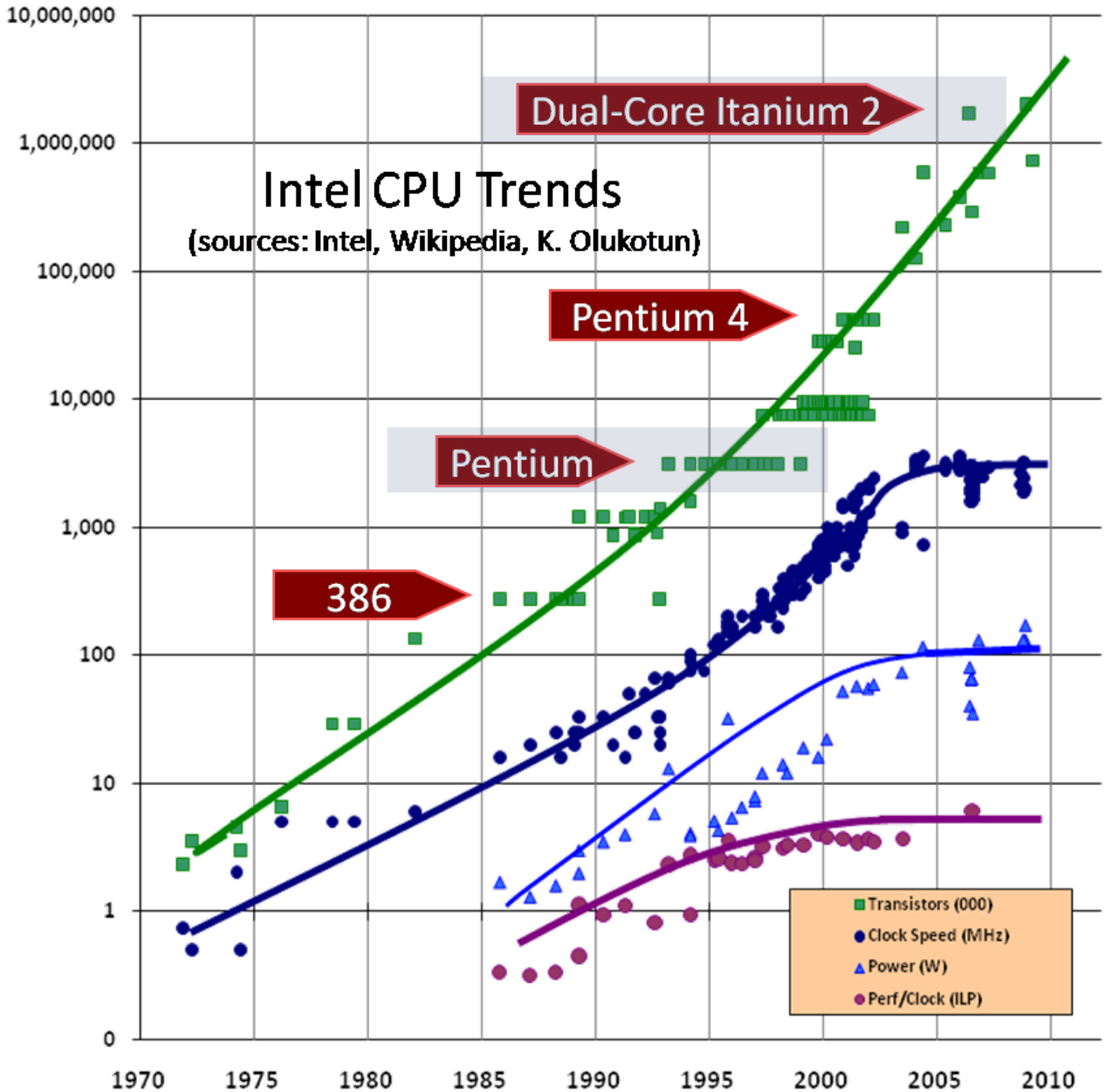
Heute gibt es:

- Einen fundamenal anderen Ansatz nebenläufiger Datenmodifikation
  - Keine Locks und Conditional variables
  - Sondern: Transaktionen!
  - Software transactional memory (STM)
- Implementierung in Haskell: `atomically`, `retry`, `orElse`
- Fallbeispiele:
  - Puffer: Reader-/Writer
  - Speisende Philosophen (nur im Code: `MySTM.hs`)
  - Weihnachtlich: das Santa Claus Problem

Die Nebenläufigkeitsrevolution

- Free lunch is over: CPUs werden nicht mehr schneller
  - 2GHz hatten wir schon 2001!
  - Heute: $\sim$3GHz
- Dafür vermehren sie sich plötzlich!
  - Quad-Core, Octo-Core, Cluster, ...
- Die großen Chip-Hersteller arbeiten an Mehr{Kern, CPU, Rechner}-Lösungen, um Moore's Law nicht zu verletzen
  - Intel Core™2, AMD Phenom™, nVidia GeForce Dual-Chip (SLI™)
- Auswirkungen auf Softwareentwicklung!
  - Verstärkter Fokus auf nebenläufige SW

Schneller geht's nimmer



Locks und Conditional variables

- Aktueller “Stand der Technik”
  - C

```
pthread_mutex_lock(&mutex)
pthread_mutex_unlock(&mutex)
pthread_cond_wait(&cond, &mutex)
pthread_cond_broadcast(&cond)
```

  - Haskell

```
newMVar :: a → IO (MVar a)
takeMVar :: MVar a → IO a
putMVar :: MVar a → a → IO ()
```

  - Java

```
synchronized public void workOnSharedData() { ... }
```

Locks und Conditional variables (2)

- Grundlegende Idee: Zugriff auf gemeinsame Ressourcen nur innerhalb kritischer Abschnitte
  1. Vor Betreten um Erlaubnis fragen (Lock an sich reißen)
  2. Arbeiten
  3. Beim Verlassen Meldung machen (Lock freigeben)

Locks und Conditional variables (2)

- Grundlegende Idee: Zugriff auf gemeinsame Ressourcen nur innerhalb kritischer Abschnitte
  1. Vor Betreten um Erlaubnis fragen (Lock an sich reißen)
  2. Arbeiten
  3. Beim Verlassen Meldung machen (Lock freigeben)
- Verfeinerung: Auf Eintreten von Bedingungen warten (Kommunikation)
  1. Im kritischen Abschnitt schlafengehen, wenn Bedingung nicht erfüllt (Lock freigeben!)
  2. Andere Threads machen Bedingung wahr und melden dies
  3. Sobald Lock verfügbar: aufwachen

Locks und Conditional variables (2)

- Grundlegende Idee: Zugriff auf gemeinsame Ressourcen nur innerhalb kritischer Abschnitte
  1. Vor Betreten um Erlaubnis fragen (Lock an sich reißen)
  2. Arbeiten
  3. Beim Verlassen Meldung machen (Lock freigeben)
- Verfeinerung: Auf Eintreten von Bedingungen warten (Kommunikation)
  1. Im kritischen Abschnitt schlafengehen, wenn Bedingung nicht erfüllt (Lock freigeben!)
  2. Andere Threads machen Bedingung wahr und melden dies
  3. Sobald Lock verfügbar: aufwachen
- Semaphoren & Monitore bauen essentiell auf demselben Prinzip auf

Kritik am Lock-basierten Ansatz

- Kritische Abschnitte haben eine pessimistische Lebenseinstellung:
  - Möglicherweise will ein anderer Thread gerade dieselben Daten verändern
  - Darum: Sperrung des Abschnitts in jedem Fall
  - Möglicherweise gar nicht nötig: Effizienz?

Kritik am Lock-basierten Ansatz

- Kritische Abschnitte haben eine pessimistische Lebenseinstellung:
  - Möglicherweise will ein anderer Thread gerade dieselben Daten verändern
  - Darum: Sperrung des Abschnitts in jedem Fall
  - Möglicherweise gar nicht nötig: Effizienz?
- Gefahr des Deadlocks:
  - A betritt kritischen Abschnitt S_1; gleichzeitig betritt B S_2
  - A will nun S_2 betreten, während es Lock für S_1 hält
  - B will dasselbe mit S_1 tun.
  - The rest is silence...

Kritik am Lock-basierten Ansatz

- Kritische Abschnitte haben eine pessimistische Lebenseinstellung:
  - Möglicherweise will ein anderer Thread gerade dieselben Daten verändern
  - Darum: Sperrung des Abschnitts in jedem Fall
  - Möglicherweise gar nicht nötig: Effizienz?
- Gefahr des Deadlocks:
  - A betritt kritischen Abschnitt S_1; gleichzeitig betritt B S_2
  - A will nun S_2 betreten, während es Lock für S_1 hält
  - B will dasselbe mit S_1 tun.
  - The rest is silence...
- Richtige Granularität schwer zu bestimmen
  - Grobkörnig: ineffizient; feinkörnig: schwer zu analysieren

Kritik am Lock-basierten Ansatz (2)

- Größtes Problem jedoch: *Lock-basierte Programme sind nicht komponierbar!*
  - Korrekte Einzelbausteine können zu fehlerhaften Programmen zusammengesetzt werden
- Klassisches Beispiel: Übertrag eines Elements von einem Container in einen anderen
  - Container-Bücherei explizit thread-safe, d.h. nebenläufiger Zugriff sicher
  - Implementierung des Übertrags:

```
transferItem item c1 c2 = do
  removeFrom c1 item
  insertInto c2 item
```

  - Problem: Zwischenzustand, in dem item in keinem Container ist
  - Plötzlich doch wieder Container-Locks erforderlich! Welche?

Kritik am Lock-basierten Ansatz (3)

- Ein ähnliches Argument gilt für Komposition von Ressourcen-Auswahl:
- Mehrfachauswahl in Unix/Linux/Mac OS X:
  - select () wartet auf mehrere I/O-Kanäle gleichzeitig
  - Kehrt zurück sobald mindestens einer verfügbar
- Annahme: Prozeduren foo() und bar() warten auf unterschiedliche Ressourcen(-Mengen). Z. B.

```
void foo(void) { /* ... */
  select(k, rd, wd, ed, &timeout);
}
```

- Keine Möglichkeit, foo() und bar() zu komponieren, etwa select (&foo, &bar)

Software transactional memory (atomicity)

- Ein optimistischer Ansatz zur Nebenläufigen Programmierung
- Prinzip der Transaktionen aus Datenbank-Domäne entliehen
- Kernidee: atomically (...) Blöcke werden atomar ausgeführt
  - (Speicher-)Änderungen erfolgen entweder vollständig oder gar nicht
  - Im letzteren Fall: Wiederholung der Ausführung
  - Im Block: konsistente Sicht auf Speicher
  - A(tomicity) und I(solation) aus ACID
- Damit deklarative Formulierung des Elementtransfers möglich:

```
atomically $
  do { removeFrom c1 item; insertInto c2 item }
```

Gedankenmodell für atomare Speicheränderungen

Mögliche Implementierung

- Thread T_1 im `atomically`-Block nimmt keine Speicheränderungen vor, sondern in schreibt Lese-/Schreiboperationen in Transaktions-Log
- Leseoperationen konsultieren zunächst Log
- Beim Verlassen des `atomically`-Blocks:
  1. globales Lock greifen
  2. konsistenter Speicher gelesen?
  3t. Änderungen einpflegen
  4t. Lock freigeben
  3f. Änderungen verwerfen
  4f. Lock freigeben, Block wiederholen

Gedankenmodell für atomare Speicheränderungen

Mögliche Implementierung

- Thread T_1 im `atomically`-Block nimmt keine Speicheränderungen vor, sondern in schreibt Lese-/Schreiboperationen in Transaktions-Log
- Leseoperationen konsultieren zunächst Log
- Beim Verlassen des `atomically`-Blocks:
  1. globales Lock greifen
  2. konsistenter Speicher gelesen?
  3t. Änderungen einpflegen
  4t. Lock freigeben
  3f. Änderungen verwerfen
  4f. Lock freigeben, Block wiederholen

Konsistenter Speicher

- Jede zugegriffene Speicherstelle hat zum Prüfzeitpunkt denselben Wert wie beim ersten Lesen

Beispiel: Konsistenter Speicher

```
atomically $
    do b1 ← getBalance acc1
       b2 ← getBalance acc2
       if sum [b1, b2] ≥ amnt
         then withdrawMoney amnt else ...
```

- Mögliches Problem: anderer Thread ruft withdrawMoney zwischen getBalance-Aufrufen auf
- acc1/2 nicht involviert: egal
- acc1 geändert
  - Wird bei Prüfung aller gelesenen/geschriebenen Speicherstellen bemerkt
- acc2 geändert
  - Egal! Konsistenz bezieht sich dann auf neuen Zustand

Blockieren / Warten (blocking)

- Atomarität allein reicht nicht: STM muss Synchronisation von Threads ermöglichen
- Klassisches Beispiel: Produzenten + Konsumenten:
  - Wo nichts ist, kann nichts konsumiert werden
  - Konsument wartet auf Ergebnisse des Produzenten

```
consumer buf = do
  item ← getItem buf
  doSomethingWith item
```

  - getItem sollte blockieren, wenn keine Items verfügbar

Kompositionales “Blockieren” mit `retry`

- Idee: ist notwendige Bedingung innerhalb atomically nicht erfüllt, wird Transaktion abgebrochen und erneut versucht

```
atomically $ do
  ...
  if (Buffer.empty buf) then retry else ...
```

- Sinnlos, sofern andere Threads Zustand nicht verändert haben!
- Daher: warten (auf Godot?)
  - Auf Änderung an in Transaktion gelesenen Variablen!
  - Genial: System verantwortlich für Verwaltung der Aufweckbedingung
- Keine lost wakeups, keine händische Verwaltung von Conditional variables

Auswahl (choice)

- Dritte Zutat für erfolgreiches kompositionales Multithreading: Auswahl möglicher Aktionen
- Beispiel: Event-basierter Webserver liest Daten von mehreren Verbindungen
- Kombinator orElse ermöglicht linksorientierte Auswahl (ähnlich $||$):

```
webServer = do
  ...
  news ← atomically $ orElse spiegelRSS cnnRSS
  req ← atomically $ foldr1 orElse clients
  ...
```

- Wenn linke Transaktion misslingt, wird rechte Transaktion versucht

Einschränkungen an Transaktionen

- Transaktionen dürfen nicht beliebige Seiteneffekte haben
  - Nicht jeder reale Seiteneffekt lässt sich rückgängig machen
  - **if** allElseFails **then** blackmailBossWith nudePics **else** ...
  - Seiteneffekte auf Transaktionsspeicher zu beschränken
- Ideal: Trennung statisch erzwingen
  - In Haskell: Trennung im Typsystem
  - IO-Aktionen und STM-Aktionen (Monaden)
  - STM Monade erlaubt Erzeugung und Verwendung von Referenzen (ähnlich MVar, IORef)

STM in Haskell

```
newtype STM a
instance Monad STM
atomically :: STM a → IO a
retry :: STM a
orElse :: STM a → STM a → STM a

data TVar
newTVar :: a → STM (TVar a)
readTVar :: TVar a → STM a
writeTVar :: TVar a → a → STM ()
```

Puffer mit STM: Modul `MyBuffer`

- Erzeugen eines neuen Puffers: newTVar mit leerer Liste

```
newtype Buf a = B (TVar [a])

new :: STM (Buf a)
new = do tv ← newTVar []
```

Puffer mit STM: Modul `MyBuffer` (2)

- Elemente zum Puffer hinzufügen (immer möglich):
  - Puffer lesen
  - Element hinten anhängen
  - Puffer schreiben

```
put :: Buf a → a → STM ()
put (B tv) x = do xs ← readTVar tv
```

Puffer mit STM: Modul `MyBuffer` (3)

- Element herausnehmen: Möglicherweise keine Elemente vorhanden!
  - Wenn kein Element da, wiederholen
  - Ansonsten: Element entnehmen, Puffer verkleinern

```
get :: Buf a → STM a
get (B tv) = do xs ← readTVar tv
                case xs of
                  [] → retry
                  (y:xs') → do writeTVar tv xs'
```

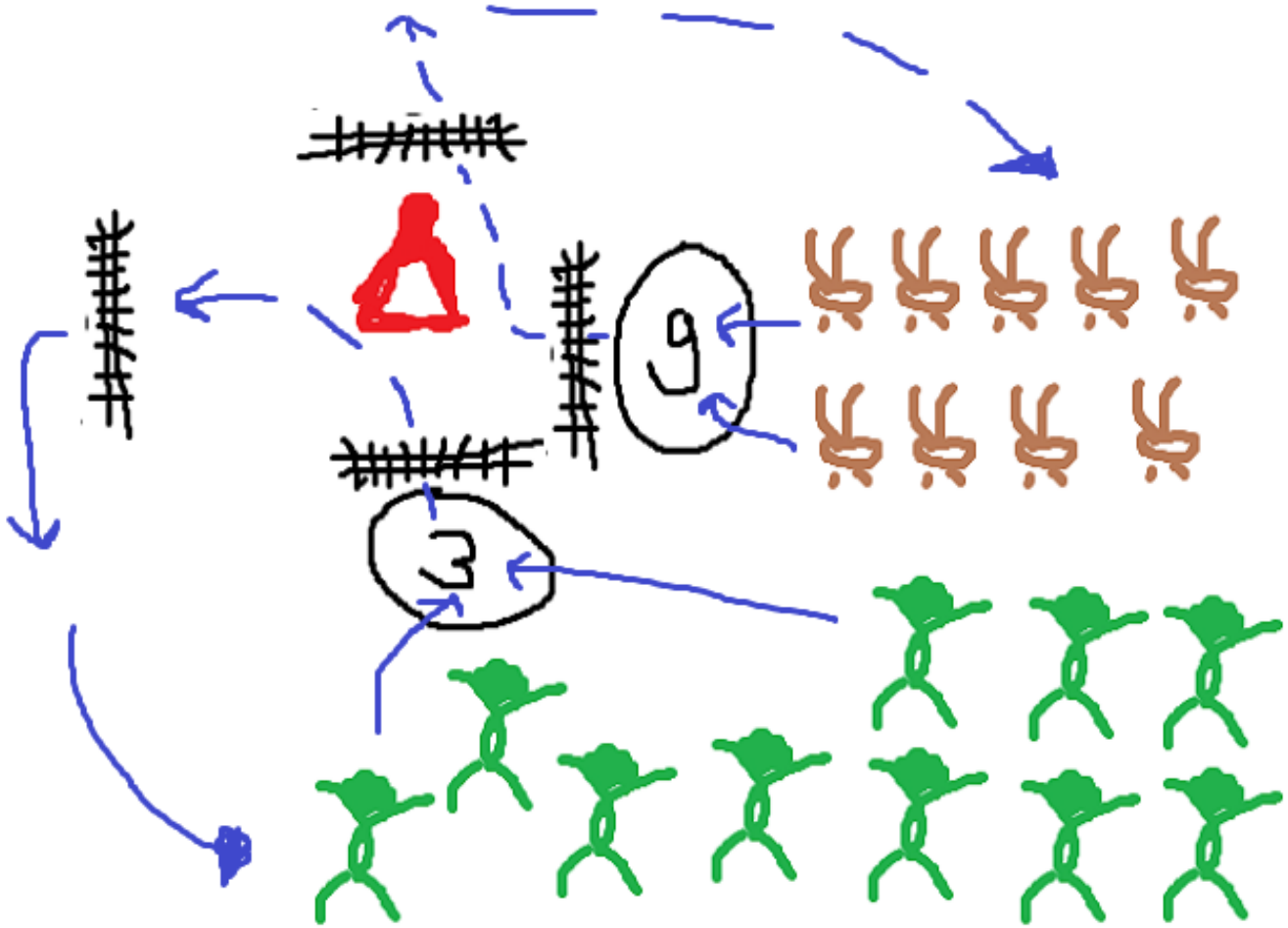
Puffer mit STM: Anwendungsbeispiel

```
useBuffer = do
  b ← atomically $ new
  forkIO $ forever $
    do threadDelay (10^6)
       t ← Tm.getClockTime >>= Tm.toCalendarTime
       atomically $ put b $
         Tm.calendarTimeToString t
  forever $ do x ← atomically $ get b
               putStrLn x
```

Santa Claus Problem

Santa repeatedly sleeps until wakened by either all of his nine reindeer, [...], or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them ([...]). If awakened by a group of elves, he shows each of the group into his study, consults with them [...], and finally shows them each out ([...]). Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting.

Santa Claus Problem, veranschaulicht



Lösungsstrategie

- Modellieren jede Elfe, jedes Rentier, jeden Weihnachtsmann als Faden
  - Santa wartet und koordiniert, sobald genügend “Teilnehmer” vorhanden
  - Elfen und Rentiere tun fortwährend dasselbe: Sammeln, arbeiten, herumstehen

Lösungsstrategie

- Modellieren jede Elfe, jedes Rentier, jeden Weihnachtsmann als Faden
  - Santa wartet und koordiniert, sobald genügend “Teilnehmer” vorhanden
  - Elfen und Rentiere tun fortwährend dasselbe: Sammeln, arbeiten, herumstehen
- Verwenden Gruppen (Group) als Sammelplätze für Elfen und Rentiere
  - 3er-Gruppe für Elfen, 9er-Gruppe für Rentiere
  - Santa wacht auf, sobald Gruppe vollzählig

Lösungsstrategie

- Modellieren jede Elfe, jedes Rentier, jeden Weihnachtsmann als Faden
  - Santa wartet und koordiniert, sobald genügend "Teilnehmer" vorhanden
  - Elfen und Rentiere tun fortwährend dasselbe: Sammeln, arbeiten, herumstehen
- Verwenden Gruppen (Group) als Sammelplätze für Elfen und Rentiere
  - 3er-Gruppe für Elfen, 9er-Gruppe für Rentiere
  - Santa wacht auf, sobald Gruppe vollzählig
- Gatterpaare (Gate) erlauben koordinierten Eintritt in Santas Reich
  - Stellt geordneten Ablauf sicher (kein Überholen übereifriger Elfen)

Vorarbeiten: (Debug-)Ausgabe der Aktionen in Puffer

```
{- Actions of elves and deer -}
meetInStudy :: Buf → Int → IO ()
meetInStudy buf id = bput buf $
  "Elf " ++ show id ++ " meeting in the study"

deliverToys :: Buf → Int → IO ()
deliverToys buf id = bput buf $
  "Reindeer " ++ show id ++ " delivering toys"
```

- Puffer wichtig, da putStrLn nicht thread-sicher!
- Lese-Thread liest Daten aus Buf und gibt sie sequentiell an stdout aus

Arbeitsablauf von Elfen und Rentieren

- Generisch: Tun im Grunde dasselbe, parametrisiert über task

```
helper1 :: Group → IO () → IO ()
helper1 grp task = do
  (inGate, outGate) ← joinGroup grp
  passGate inGate
  task
  passGate outGate

elf1, reindeer1 :: Buf → Group → Int → IO ()
elf1 buf grp elfId =
  helper1 grp (meetInStudy buf elfId)
reindeer1 buf grp reinId =
  helper1 grp (deliverToys buf reinId)
```

Gatter: Erzeugung, Durchgang

- Gatter haben aktuelle sowie Gesamtkapazität
- Anfänglich leere Aktualkapazität (Santa kontrolliert Durchgang)

```
data Gate = Gate Int (TVar Int)

newGate :: Int → STM Gate
newGate n = do tv ← newTVar 0
               return $ Gate n tv

passGate :: Gate → IO ()
passGate (Gate n tv) =
  atomically $ do c ← readTVar tv
                  check (c > 0)
                  writeTVar tv (c − 1)
```

Nützliches Design Pattern: `check`

- Nebenläufiges assert:

```
check :: Bool → STM ()
check b | b = return ()
        | not b = retry
```

  - Bedingung b muss gelten, um weiterzumachen
  - Im STM-Kontext: wenn Bedingung nicht gilt: wiederholen
  - Nach check: Annahme, dass b gilt

- Wunderschön deklarativ!

Santas Aufgabe: Gatter betätigen

- Wird ausgeführt, sobald sich eine Gruppe versammelt hat
- Zwei atomare Schritte
  - Kapazität hochsetzen auf Maximum
  - Warten, bis Aktualkapazität auf 0 gesunken ist, d.h. alle Elfen/Rentiere das Gatter passiert haben

```
operateGate :: Gate → IO ()
operateGate (Gate n tv) = do
  atomically $ writeTVar tv n
  atomically $ do c ← readTVar tv
                  check (c ≡ 0)
```

- Beachte: Mit nur einem atomically wäre diese Operation niemals ausführbar! (Starvation)

Gruppen: Erzeugung, Beitritt

```
data Group = Group Int (TVar (Int, Gate, Gate))

newGroup :: Int → IO Group
newGroup n = atomically $ do
  g1 ← newGate n
  g2 ← newGate n
  tv ← newTVar (n, g1, g2)
  return $ Group n tv

joinGroup :: Group → IO (Gate, Gate)
joinGroup (Group n tv) =
  atomically $ do (k, g1, g2) ← readTVar tv
                  check (k > 0)
                  writeTVar tv (k − 1, g1, g2)
                  return $ (g1, g2)
```

joinGroup

- Noch einmal zum Staunen:

```
atomically $ do (k, g1, g2) ← readTVar tv
                check (k > 0)
                writeTVar tv (k − 1, g1, g2)
                return $ (g1, g2)
```

- Nebenläufigkeit ist also schwierig?!

Eine Gruppe erwarten

- Santa erwartet Elfen und Rentiere in entspr. Gruppengröße
- Erzeugt neue Gatter für nächsten Rutsch
  - Verhindert, dass Elfen/Rene

```
awaitGroup :: Group → STM (Gate, Gate)
awaitGroup (Group n tv) = do
  (k, g1, g2) ← readTVar tv
  check (k ≡ 0)
  g1' ← newGate n
  g2' ← newGate n
  writeTVar tv (n, g1', g2')
  return (g1, g2)
```

Elfen und Rentiere

- Für jeden Elf und jedes Rentier wird ein eigener Thread erzeugt
- Bereits gezeigte elf1, reindeer1, gefolgt von Verzögerung (für nachvollziehbare Ausgabe)

```
-- An elf does his elf thing, indefinitely.
elf :: Buf → Group → Int → IO ThreadId
elf buf grp id = forkIO $ forever $
  do elf1 buf grp id
     randomDelay

-- So does a deer.
reindeer :: Buf → Group → Int → IO ThreadId
reindeer buf grp id = forkIO $ forever $
  do reindeer1 buf grp id
     randomDelay
```

Santa Claus' Arbeitsablauf

- Gruppe auswählen, Eingangsgatter öffnen, Ausgang öffnen
- Zur Erinnerung: operateGate "blockiert", bis alle Gruppenmitglieder Gatter durchschritten haben

```
santa :: Buf → Group → Group → IO ()
santa buf elves deer = do
  (name, (g1, g2)) ← atomically $
    chooseGroup "reindeer" deer `orElse`
      chooseGroup "elves" elves
  bput buf $ "Ho, ho, my dear " ++ name
  operateGate g1
  operateGate g2

chooseGroup :: String → Group →
               STM (String, (Gate, Gate))
chooseGroup msg grp = do
  gs ← awaitGroup grp
  return (msg, gs)
```

Hauptprogramm

- Gruppen erzeugen, Elfen und Rentiere “starten”, santa ausführen

```
main :: IO ()
main = do buf ← setupBufferListener

          elfGroup ← newGroup 3
          sequence_ [ elf buf elfGroup id |
                      id ← [1 .. 10] ]
          deerGroup ← newGroup 9
          sequence_ [ reindeer buf deerGroup id |
                      id ← [1 .. 9]]
          forever (santa buf elfGroup deerGroup)
```

Zusammenfassung

- *The future is now, the future is concurrent*
- Lock-basierte Nebenläufigkeitsansätze skalieren schlecht
  - Korrekte Einzelteile können nicht ohne weiteres komponiert werden
- Software Transactional Memory als Lock-freie Alternative
  - Atomarität (atomically), Blockieren (retry), Choice (orElse) als Fundamente kompositionaler Nebenläufigkeit
  - Faszinierend einfache Implementierungen gängiger Nebenläufigkeitsaufgaben
- Das freut auch den Weihnachtsmann:
  - Santa Claus Problem in STM Haskell

Literatur

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy.
Composable memory transactions.
In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

Simon Peyton Jones.
Beautiful concurrency.
In Greg Wilson, editor, *Beautiful code*. O'Reilly, 2007.

Herb Sutter.
The free lunch is over: a fundamental turn toward concurrency in software.
Dr. Dobb's Journal, 30(3), March 2005.

Fortgeschrittene Techniken der Funktionalen Programmierung

Vorlesung vom 05.01.10: Fancy Types

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
- Teil IV: The Future of Programming
  - Fancy Types
  - Domain-Specific Languages
  - The Next Big Thing: F#, Scala
  - Rückblick, Ausblick

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, dynamische Bindung
- nur Funktional: algebraische Typen, Fkt. höherer Ordnung. strengere Typisierung

Dynamische Bindung

- Java: Auflösung der Methoden zur Laufzeit:

```
class Shape
  { void draw()
    { System.out.println("*"); }}
class Triangle extends Shape
  { void draw()
    { System.out.println("/\\"); }}
class Circle extends Shape
  { void draw()
    { System.out.println("()"); }}
```

Dynamische Bindung

- Java: Auflösung der Methoden zur Laufzeit:

```
class test {
    public static void main(String [] a) {
    Shape s;
    if (a[0].equals("t")) { s= new Triangle(); }
    else if (a[0].equals("s")) { s= new Circle(); }
    else { s= new Shape(); }

    s.draw();}}
```

Dynamische Bindung in Haskell?

```
type Point = (Double, Double)

class Shape s where draw :: s → String
                    draw _ = "*"

data Triangle = Triangle Point Point Point
instance Shape Triangle where
  draw Triangle {} = "/\\"

data Circle = Circle Point Double
instance Shape Circle where
  draw Circle {} = "()"
```

- So nicht ...

Dynamische Bindung in Haskell?

- Typ wird zur Übersetzungszeit berechnet.
  - Obwohl erst zur Laufzeit gebunden!
  - Implementierung von Klassen durch dictionaries
  - Typisierung verdeckt dynamische Bindung
- Warum dynamische Bindung?
  - Vorteil: Erweiterbarkeit eingebaut
  - Nachteil: Erweiterbarkeit nicht immer erwünscht

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-Howard-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. App a (a → Int)
```

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

Ein einfaches Beispiel

- Ein existenzieller Typ:

```
data T = forall a. App a (a → Int)

ap :: T → Int
ap (App x f) = f x
```

- Anwendung:

```
  map ap [App [1,2,3] length,
          App 3 (5 +),
          App getLine (λ_ → 0)]
```

- Nützlich?
- Benötigt: Signatur für a

Heterogen Listen

- Signaturen im großen: Module
- Signaturen im kleinen: Typklassen
- Klassen für Typvariablen

```
data S = forall a. Show a⇒ Cons a (a→ Int)

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

Dynamische Bindung in Haskell?

- Beispiel: Shapes revisited

```
class Shape s where draw :: s → String
                    draw _ = "*"
```

- Damit heterogene Listen von Shapes (selbstgemacht)

```
data ShapeList =
     forall s. Shape s⇒ Cons s ShapeList
   | Empty
```

- Obertyp aller Shapes:

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

Dynamische Bindung in Haskell?

- Dreiecke und Kreise:

```
data Triangle = Triangle Point Point Point
instance Shape Triangle where draw Triangle {} = "/\

data Circle = Circle Point Double
instance Shape Circle where draw Circle {} = "()"
```

- Davon unabhängig Quadrate:

```
data Square = Square Point Point
instance Shape Square where draw Square {} = "[]"
```

- Zusammenfügen:

```
shapelist1 = [Shape tri, Shape sq, Shape circle]
shapelist2 = Cons tri (Cons sq (Cons circle Empty))
shapelist2' = foldr (λ (Shape x) l → Cons x l) Empty
```

Dynamische Bindung in Haskell?

- Auflösung der Bindung zur Laufzeit
- Damit: heterogene Datenstrukturen
- Keine echte Vererbung
- 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.
- 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
```

  - Allquantor links $\cong$ Existenzquantor

- 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 $(* \rightarrow *) \rightarrow *$
- Monomorphe Typen haben Kind *
- Polymorphe Typen haben Kind $* \rightarrow *$
- 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 etc Typüberprüfung unentscheidbar.
  - D.h. Typcheck kann divergieren!
- Aber: wen kümmert's?
- Hindley-Milner (Haskell) ist exponentiell.
  - Kann so gut wie divergieren
  - Beispiel

Zusammenfassung

- "Abstract types have existential type"
- (Limitierte) Modellierung von Objektorientierung
  - Keine Vererbung, Erweiterbarkeit
- Rang-n Polymorphie
  - Beispiele: `runST`, generische Programmierung, ...
- Typüberprüfung wird unentscheidbar
  - ... aber schon Hindley-Milner-Typcheck ist exponentiell!
- Nächste Woche: Keine Vorlesung!
- Danach: DSLs (Domain-Specific Languages)

Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 19.01.10:
Domain Specific Languages (DSLs)

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
- Teil IV: The Future of Programming
  - Fancy Types
  - Domain-Specific Languages
  - The Next Big Thing: F#, Scala
  - Rückblick, Ausblick

Themen

- Domain specific languages: Definition & Einordnung
- Eingebettete DSLs in Haskell
  - Einbettung: Vor-/Nachteile
  - Kompilierung vs. Interpretation
  - Tiefe vs. flache Einbettung
  - Repräsentation von Graphen
- Beispiele
  - Statecharts
  - Haskell XML Toolkit
  - Lava Hardware Description Language

DSLs contra allgemeine Programmiersprache

- Bei (informatischen) Problemstellungen gibt es oft zwei Möglichkeiten
  - eine spezifische Lösung, die das konkrete Problem sehr effizient/elegant/kostensparend löst, aber nicht/schwer auf andere/ggf. ähnliche Probleme anwendbar ist
  - eine generische Lösung, die auch andere Probleme abdeckt/abdecken kann, aber mehr Anpassungsaufwand erfordert/weniger effizient bei der Lösung des vorliegenden Problems ist
- Diese Dichotomie spiegelt sich in der Unterscheidung zwischen allgemeinen Programmiersprachen (general purpose languages, GPL) und domänenspezifischen Sprachen (DSL) wider
  - Lösen einer großen Klasse von Problemen durch Bereitstellen einer Bücherei in einer (meist Turing-mächtigen) GPL
  - Lösen einer wohldefinierten Unterklasse von Problemen mit einer DSL

Allgemeine Programmiersprache

- Das Konzept der domänenspezifischen Ausrichtung trifft im Grunde auch auf klassiche Programmiersprachen zu:
  - Cobol wurde für die Abbildung von Geschäftsprozessen entworfen
  - Fortran: numerische Berechnungen (“number crunching”)
  - Lisp: Symbolische Berechnungen (v. a. in der KI)
  - Erlang: Telecommunication Switching, fehlertolerante nebenläufige Systeme
- Aber: diese wurden über die Zeit zu allgemeinen Programmiersprachen erweitert
- Wiederum ein generelles Phänomen: Sprachen tendieren dazu, “fett” zu werden.
  - Keine Sprache ist zur Entwurfszeit perfekt
  - Anwender fordern neue Features, die sie aus anderen Sprachen kennen
  - Neuentwurf/Reduktion einer Sprache nicht praktikabel, daher meist monotones Wachstum der bestehenden Sprache

DSL: Definition 1

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

(van Deursen et al., 2000)

DSL: Definition 2

In software development, a domain-specific language (DSL) is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. The concept isn't new – special-purpose programming languages and all kinds of modeling/specification languages have always existed –, but the term has become more popular due to the rise of domain-specific modeling.

(Wikipedia, 2010-01-18)

Eigenschaften von DSLs

- Fokussierte Ausdrucksmächtigkeit
  - Turing-Mächtigkeit kein Ausschlusskriterium, aber auch nicht Ziel der Sprache.
  - Oftmals deutlich weniger mächtig: Reguläre Ausdrücke, Makefiles,
- DSLs sind üblicherweise klein (daher auch die Bezeichnungen “little language” und “micro-language”), d. h. die Anzahl der Sprachkonstrukte ist eingeschränkt, aber auf die Anwendung zugeschnitten
- Meist sind sie deklarativ statt imperativ: XSLT, Relax NG Schemas, Graphviz/Dot, Excel Formeln. . .
- Spiegeln in Sprachkonstrukten und Vokabular die Domäne wider

Weitere Abgrenzung

Programmierschnittstellen (APIs)

- Etwa jUnit: `assertTrue()`, `assertEquals()` Methoden & `@Before`, `@Test`, `@After` Annotationen
- Funktionsnamen spiegeln ebenfalls Domänenvokabular wider
- Gängige Sprachen (Java, C/C++, Ada) erschweren weitere Abstraktion: Syntaxerweiterungen, Konzepte höherer Ordnung
- Imperative Ausrichtung der Programmiersprache vs. deklarative DSL

Weitere Abgrenzung

Programmierschnittstellen (APIs)

- Etwa jUnit: `assertTrue()`, `assertEquals()` Methoden & `@Before`, `@Test`, `@After` Annotationen
- Funktionsnamen spiegeln ebenfalls Domänenvokabular wider
- Gängige Sprachen (Java, C/C++, Ada) erschweren weitere Abstraktion: Syntaxerweiterungen, Konzepte höherer Ordnung
- Imperative Ausrichtung der Programmiersprache vs. deklarative DSL

Skriptsprachen

- JavaScript, Lua, Ruby werden für DS-artige Aufgaben verwendet
  - HTML/XML DOM-Manipulation
  - Game Scripting (z. B. mit Lua in World of Warcraft)
  - Webprogrammierung: Ruby on Rails
- Grundausrichtung: programmatische Erweiterung von Systemen

DSL-Beispiel: Relax NG

Adressbuchformat

```
grammar {
  start = entries
  entries = element entries { entry* }
  entry = element entry {
    attribute name { text },
    attribute birth { xsd:dateTime },
    text }
}
```

- Beschreibung von XML-Bäumen
  - Erlaubte Element-Verschachtelungen & -Reihenfolgen
  - Datentypen von Attributen & Elementwerten
- Mögliche automatische Generierung von Validatoren

DSL-Beispiel: Reguläre Ausdrücke

Textsuche und -ersetzung

```
sed -e 's/href=\(\"[^\"]*\"\)/src=\1/g'

egrep '^\[.*\]$'
```

- Extrem effiziente Implementierung als endliche Automaten
- Konzise, wenn auch unleserliche, Beschreibung von Textmustern
- Elementarer Bestandteil von Skriptsprachen (Perl, Python, JavaScript...)

DSL-Beispiel: VHDL

```
ENTITY DFlipflop IS
    PORT(D,Clk: IN Bit;
         Q: OUT Bit);
  END DFlipflop;
  ARCHITECTURE Behav OF DFlipflop IS
      CONSTANT T_Clk_Q: time := 4.23 ns;
  BEGIN
      PROCESS
      BEGIN
         WAIT UNTIL Clk'EVENT AND Clk'Last_Value='0' AND
                                                Clk='1';
         Q<=D AFTER T_Clk_Q;
      END PROCESS;
  END Behav;
```

Vorteile der Verwendung von DSLs

- Ausdruck von Problemen/Lösungen in der Sprache und auf dem Abstraktionslevel der Anwendungsdomäne
- Notation matters: Programmiersprachen bieten oftmals nicht die Möglichkeit, Konstrukte der Domäne angemessen wiederzugeben
- DSL-Lösungen sind oftmals selbstdokumentierend und knapp
- Bessere (automatische) Analyse, Optimierung und Testfallgenerierung von Programmen
  - Klar umrissene Domänensemantik
  - eingeschränkte Sprachmächtigkeit $\Rightarrow$ weniger Berechenbarkeitsfallen
- Leichter von Nicht-Programmierern zu erlernen als GPLs

Nachteile der Verwendung von DSLs

- Hohe initiale Entwicklungskosten
- Schulungsbedarf
- Sprachdesign ist eine äußerst schwierige und komplexe Angelegenheit, deren Aufwand nahezu immer unterschätzt wird
- Fehlender Tool-Support
  - Debugger
  - Generierung von (Online-)Dokumentation
  - Statische Analysen, ...
- Effizienz: Interpretation ggf. langsamer als direkte Implementierung in GPL

Auswertung von DSLs

- Kompilierung: Maßgeschneiderter Code für ein gegebenes DSL-Programm wird erzeugt (und ggf. übersetzt)
  - *Lex/Yacc* erzeugen C Code für spezifischen Lexer (insb. Zustandsautomat) und Parser
  - *Pan* (Functional Images, C. Elliott) übersetzt Haskell Datentypen in C Programme, die Bilddateien bzw. Bildanzeige generieren
- Interpretation mit tiefer Einbettung
  - Programme als Datentypen (`Expr`) der interpretierenden Sprache (host language)
  - Interpretationsfunktion `I : Expr -> Env -> Value` evaluiert Programm und Eingaben zu Wert der Domäne (+ Ausgaben)
- Interpretation mit flacher Einbettung
  - DSL-Operatoren arbeiten direkt auf semantischer Domäne
  - `Add : Value -> Value -> Value`

Domain-specific embedded languages

- Um Probleme mit Sprachdefinition und Interpreterimplementierung zu vermeiden, kann eine DSL auch direkt in eine GPL eingebettet werden
  - Vorhandenes Ausführungsmodell und Werkzeuge
- Funktionale Sprachen eignen sich hierfür besonders gut
  - Algebraische Datentypen zur Termrepräsentation
  - Funktional $\subseteq$ Deklarativ
  - Funktionen höherer Ordnung ideal für Kombinatoren
  - Interpreter (`ghci`, `ocaml`, ...) erlauben mit sich entwickelnder DSL herumzuspielen

XML-Verarbeitung mit dem Haskell XML Toolkit (HXT)

- Eine in Haskell eingebettete DSL zur Verarbeitung von XML-Dokumenten
  - Eingabe/Ausgabe von XML
  - Transformationen auf XML DOM
- Kann als XPath-Ersatz dienen (zur Auswahl von Bestandteilen eines XML-Dokuments); durch Einsatz von beliebigen Haskell-Code jedoch deutlich mächtiger
- Ebenfalls möglich: Einsatz als XSLT-Ersatz: Transformation von einem XML-Schema in ein anderes
- Kombinator-basiert: ein fester Satz von unären und binären Haskell-Funktionen repräsentiert die Sprache zur Beschreibung von XML *Filtern*

HXT: Datentypen für XML-Dokumente

```
data NTree a   = NTree a [NTree a]   -- rose tree

data XNode     = XText String   -- plain text node
               | ...
               | XTag QName XmlTrees
                 -- element name and list of attributes
               | XAttr QName   -- attribute name
               | ...

type QName     = ...   -- qualified name

type XmlTree   = NTree XNode
```

HXT: XML Filter

- Elementare Datenstruktur: Mehrwertige Funktion über XmlTree

```
type XmlFilter = XmlTree → [XmlTree]

type Filter a b = a → [b]

isA  :: (a → Bool) → (a → [a])
isA p x
  | p x       = [x]
  | otherwise = []

isXText                              :: XmlFilter
isXText t@(NTree (XText _) _) =  [t]
isXText _                            =  []
```

HXT: Transformationen & partielle Fkt.

```
trans    :: XmlTree → XmlTree
trans t = exp(t)

ftrans   :: XmlTree → [XmlTree]
ftrans t = [exp(t)]

part     :: XmlTree → XmlTree
part t
  | p t =  expr(t)
  | otherwise = error "f␣not␣defined"

fpart    :: XmlFilter
fpart t
  | p t = [expr(t)]
  | otherwise = []
```

HXT: Komposition von Filtern

- Zwei Filter nacheinander ausführen:

```
(>>>)       :: XmlFilter → XmlFilter → XmlFilter
(f >>> g) t =  concat [g t' | t' ← f t]
```

- Ergebnisse zweier Filter zusammentun:

```
(<+>)       :: XmlFilter → XmlFilter → XmlFilter
(f <+> g) t =  f t ++ g t
```

HXT: Auswahl

```
orElse  :: XmlFilter → XmlFilter → XmlFilter
orElse f g t
  | null res1 = g t
  | otherwise = res1
  where res1 = f t

when    :: XmlFilter → XmlFilter → XmlFilter
when f g t
  | null (g t) = [t]
  | otherwise  = f t

guards  :: XmlFilter → XmlFilter → XmlFilter
guards g f t
  | null (g t) = []
  | otherwise  = f t
```

HXT: Traversion des XML-Baumes

- Auswahl aller Nachfahren, die f erfüllen, wobei Kinder von derlei Elementen nicht weiter untersucht werden

```
deep     :: XmlFilter → XmlFilter
deep f   = f `orElse` (getChildren ⋙ deep f)
```

- Auswahl wirklich aller Nachfahren, die f erfüllen

```
multi    :: XmlFilter → XmlFilter
multi f  = f <+> (getChildren ⋙ multi f)
```

- Weitere Beispiele: `examples-11/Hxt.hs`

HXT: Das Kleingedruckte

- HXT baut als Bücherei auf dem Konzept der *Arrows* auf
- Siehe `http://www.haskell.org/arrows/`

```
class Category cat where
  id :: cat a a
  (.) :: cat b c → cat a b → cat a c
(<<<) :: Category c ⇒ c b d → c a b → c a d
(>>>) :: Category c ⇒ c a b → c b d → c a d

class Category a ⇒ Arrow a where
arr :: (b → c) → a b c
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')
```

And now for something completely different...

Lava: Hardware Description Language

- In Haskell eingebettete Sprache zur Beschreibung von Schaltkreisen
- Industrieller Einsatz: Xilinx Inc. (S. Singh & P. Bjesse)
- In Universität Chalmers, Göteborg zur Lehre von Design und Verifikation von Hardware eingesetzt
- Dokumentation:
  `http://www.cs.chalmers.se/~koen/Lava/tutorial.ps`

- Wir bauen die wichtigsten Bestandteile hier und jetzt *live* nach!
- Siehe `examples-11/Lava.hs`

Literatur


Koen Claessen and David Sands.
Observable sharing for functional circuit description.
In P. S. Thiagarajan and R. Yap, editors, *Advances in Computing Science – ASIAN'99*, volume 1742 of *LNCS*, pages 62–73, 1999.


Paul Hudak.
Building domain-specific embedded languages.
ACM Comput. Surv., 28, 1996.


Marjan Mernik, Jan Heering, and Anthony M. Sloane.
When and how to develop domain-specific languages.
ACM Comput. Surv., 37(4):316–344, 2005.


Arie van Deursen, Paul Klint, and Joost Visser.
Domain-specific languages: an annotated bibliography.
SIGPLAN Not., 35(6):26–36, 2000.

Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 26.01.10:
The Next Big Thing — Scala & F#

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
- Teil IV: The Future of Programming
  - Fancy Types
  - Domain-Specific Languages
  - The Next Big Thing: F#, Scala
  - Rückblick, Ausblick

Heute

- Scala
- F#
- . . . warum?

Scala

- A scalable language
- Multi-paradigma-Sprache
- “Lebt im Java-Ökosystem”
- Martin Odersky, ETH Lausanne
- `http://www.scala-lang.org/`

Scala — Die Sprache

- Objekt-orientiert:
  - Veränderlicher, gekapselter Zustand
  - Subtypen und Vererbung
  - Klassen und Objekte
- Funktional:
  - Unveränderliche Werte
  - Polymorphie
  - Funktionen höherer Ordnung

Scala — Weitere Features

- `for` für Komprehension
- Abstraktionsmechanismen für Typen, Klassen, Parameter
- Unterstützt funktionale Konzepte (Endrekursion, pattern matching)
- Abbildung von Java-Konzepten: singleton objects, traits
- Flexibles, mächtiges Typsystem (Varianz, beschränkte Polymorphie)
- XML-Unterstützung

Fortgeschrittene Techniken der Funktionalen Programmierung

Vorlesung vom 02.02.10: Rückblick & Ausblick

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
- Teil IV: The Future of Programming
  - Fancy Types
  - Domain-Specific Languages
  - The Next Big Thing: F#, Scala
  - Rückblick, Ausblick

Monaden

- Monade: Einbettung imperativer Konzepte in funktionaler Sprache
- Zustandsmonade `ST s a = s -> (a, s)`
  - Lesemonade `Read r a = r -> a`
  - Schreibmonade `Writer w a = (a, [w])`
- Ausnahmemonade `Error e a = Left e | Right a`
- Nichtdeterminismus `ND a = [a]`
- IO-Aktionen sind keine Magie
- Monadentransformer: Kombination von Monaden

Fortgeschrittene Datenstrukturen

- Der Zipper: in-place-update für funktionale Sprachen
- Unendliche Datenstrukturen
- Graphen: induktiv (erzeugt), aber nicht frei (kein **data**)

Nebenläufigkeit

- Leichtgewichtige Threads in Haskell
- Kleines Basiskonzept (threads, MVar)
- Darüber Abstraktionen: Semaphoren, Monitore, Kanäle, Aktoren, ...
- Software Transactional Memory
  - Fundamental anderes Synchronisationskonzept
  - Auch in anderen Programmiersprachen interessant!

Fancy Types

- Hindley-Milner-Polymorphie: Typcheck ist exponentiell
- Abstract Types have Existential Type
- Polymorphie höherer Ordnung
  - Beispiel Zustandsübergangsmonade:
    runST :: **forall** a. (**forall** s. ST s a)→ a
- Typentheorie — der mathematische Hintergrund
- Lektüre: Cardelli/Wegner, *n Understanding Types, Data Abstraction and Polymorphism* (Klassiker!)

Domain-Specific Languages

- DSL: Programmiersprache für spezifische Problemstellung
- Beispiele: XSLT, Postscript, LaTeX, VHDL, UML, ...
- In Haskell eingebettet:
  - Tief: Programme als Datentyp
  - Flach: Programme als Haskell-Funktionen
  - Beispiele: HXT, Lava

The Next Big Thing

- Scala, F#
- These: Funktional gut für nebenläufige Programme
- Weitere Aspekte:
  - Polytypische Programmierung
  - Reflektion
  - Generic Programming
  - Template Haskell
  - Beispiel: XML Serializing

Nicht Behandelt

- Implementationsaspekte — wie übersetzt man funktionale Sprachen?
  - Meist mehrere Phasen (volle Sprache $\longrightarrow$ einfache Sprache $\longrightarrow$ ausführbare Sprache)
  - Abstrakte Ausführungsmaschinen (CAM, spineless tagless G-Machine)
- Sprachinteroperabilität
  - From Heaven to Hell and back: call-out, call-in, das Foreign Function Interface
  - component frameworks, eg. CORBA, Java, .Net
- Semantik — was bedeutet das alles?
  - Fixpunkte — Theorie der kontinuierlichen Funktionen
  - Datentypen — Domänentheorie (CPOs, $D = D \longrightarrow D$)