Vorlesung vom 31.01.2005: Roboter!

In eigener Sache

Studentische Hilfskräfte gesucht!

Die AG Krieg-Brückner sucht Haskell-Programmierer!

Tätigkeitsbereich: Werkzeugentwicklung

- Formale Programmentwicklung: `http://www.tzi.de/cofi/hets`
- Benutzerschnittstellen: `http://proofgeneral.inf.ed.ac.uk/kit`

Meldet Euch bei Till Mossakowski (`till@tzi.de`) oder mir.

Inhalt

- Modellierung von Zuständen
- Zustandsübergänge als Monaden
- Eingebettete Sprachen
- Beispiel: die Roboterkontrollsprache IRL
  - Nach Hudak, Kapitel 19.

Modellierung von Aktionen

- Aktionen (`IO a`) sind keine schwarze Magie.
- Grundprinzip:
  - Der Systemzustand wird durch das Programm gereicht.
  - Darf dabei nie dupliziert oder vergessen werden.
  - Auswertungsreihenfolge muß erhalten bleiben.
- $\Longrightarrow$ Zustandsübergangsmonaden

Zustandsübergangsmonaden

- Typ:

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

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

- `IO a` ist Instanz der Typklasse `Monad`:

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

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: Konstruktor um `ST`

Komposition von Zustandsübergängen

- Hilfsfunktion zum auspacken.

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

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

- Identität für Zustandstransformationen:

Aktionen

- Aktionen: Zustandstransformationen auf der ‘Welt’
- Typ `RealWorld#` repräsentiert Außenwelt
  - Typ hat genau einen Wert `realworld#`, der nur von initialem Aufruf erzeugt wird.
  - Aktionen: `type IO a = ST RealWorld# a`
- Optimierungen:
  - `ST s a` vordefiniert durch in-place-update implementieren;
  - `IO`-Aktionen durch einfachen Aufruf ersetzen.

IRL im Beispiel

- Alle Roboterkommandos haben Typ `Robot a`
  - `Robot a` ist Instanz von `Monad`
  - Bewegung `move :: Robot ()`, `turnLeft :: Robot ()`
  - Roboter kann zeichnen: `penUp :: Robot()`, `penDown :: Robot()`
  - Bsp: Quadrat zeichnen

```
drawSquare =
  do penDown; move; turnRight; move;
     turnRight; move; turnRight; move
```

- Roboter lebt in einer einfachen Welt mit Wänden
  - Test, ob Feld vor uns frei: `blocked :: Robot Bool`

Kontrollstrukturen

- Bedingungen und Schleifen:

```
cond  :: Robot Bool-> Robot a-> Robot a-> Robot a
cond_ :: Robot Bool-> Robot ()-> Robot ()
while :: Robot Bool-> Robot ()-> Robot ()
```

- Bsp: Ausweichen

```
evade :: Robot ()
evade = do cond_ blocked turnRight
```

- Bsp: Auf nächste Wand zufahren

```
moveToWall :: Robot ()
moveToWall = while (isnt blocked)
                   move
```

Roboter auf Schatzsuche

- Welt enhält auch Münzen.
- Münzen aufnehmen und ablegen:

```
pickCoin :: Robot (), dropCoin :: Robot ()
```

- Roboter steht auf einer Münze?

```
onCoin :: Robot Bool
```

- Beispiel: Auf dem Weg Münzen sammeln (wie `moveWall`)

```
getCoinsToWall :: Robot ()
getCoinsToWall = while (isnt blocked) $
                    do move; pickCoin
```

Implementation

- Der Roboterzustand:

```
data RobotState
 = RobotState
   { position  :: Position
   , facing    :: Direction
   , pen       :: Bool
   , color     :: Color
   , treasure  :: [Position]
   , pocket    :: Int
   } deriving Show
```

- Vergleiche `Car` (letzte Woche) — hier: abstrakter

`Robot a` als Zustandsübergang

- Erster Entwurf:

```
type Robot a = RobotState-> (RobotState, a)
```

- Aber: brauchen die Welt (`Grid`), Roboterzustände zeichnen:

```
type Robot a =
  RobotState-> Grid-> Window-> (RobotState, a, IO())
```

- Aktionen nicht erst aufsammeln, sondern gleich ausführen — `RobotState` in `IO` einbetten.

```
data Robot a =
  Robot (RobotState -> Grid -> Window -> IO (RobotState, a))
```

- Damit `Robot a` als Instanz von Monad.

Datentypen und Hilfsfunktionen

- Positionen:

```
type Position = (Int,Int)
```

- Richtungen:

```
data Direction = North | East | South | West
     deriving (Eq,Show,Enum)
```

- Drehung:

```
right,left :: Direction -> Direction
```

- Die Welt:

```
type Grid = Array Position [Direction]
```

  - Enthält für `(x,y)` Richtungen erreichbarer Nachbarfelder

Einfache Zustandsmanipulationen

- Zwei Hilfsfunktionen

```
updateState :: (RobotState -> RobotState)-> Robot ()
queryState  :: (RobotState -> a) -> Robot a
```

- Damit Implementation der einfachen Funktionen — Bsp:

```
turnLeft  :: Robot ()
turnLeft = updateState (\s -> s {facing = left (facing s)})
```

- Einzige Ausnahme: `blocked`.

Bewegung

- Beim Bewegen:
  - Prüfen, ob Bewegung möglich
  - Neue Position des Roboters zeichnen
  - Neue Position in Zustand eintragen.

```
move :: Robot ()
move = cond_ (isnt blocked) $
         Robot $ \s _ w -> do
            let newPos = movePos (position s) (facing s)
            graphicsMove w s newPos
            return (s {position = newPos}, ())
```

- Geliftete Negation: `isnt :: Robot Bool-> Robot Bool`

Grafik

- Weltkoordinaten (`Grid`): Ursprung $(0, 0)$ in Bildmitte
- Eine Position (`Position`) $\sim$ zehn Pixel
- Wände werden zwischen zwei Positionen gezeichnet
- Roboterstift zeichnet von einer Position zum nächsten
- Münzen: gelbe Kreise direkt links über der Position
- Münzen löschen durch übermalen mit schwarzem Kreis

Hauptfunktion

```
runRobot :: Robot () -> RobotState -> Grid -> IO ()
```

- Fenster öffnen
- Welt zeichnen, initiale Münzen zeichnen
- Auf Spacetaste warten
- Funktion `Robot ()` ausführen
- Aus Endzustand kleine Statistik drucken
- Zeigen.

Beispiel 1

- Roboter läuft in Spirale.
- Nach rechts drehen, n Felder laufen, nach rechts drehen, n Felder laufen;
- Dann n um eins erhöhen;
- Nützliche Hilfsfunktion:

```
for :: Int -> Robot ()-> Robot ()
for n a = sequence_ (replicate n a)
```

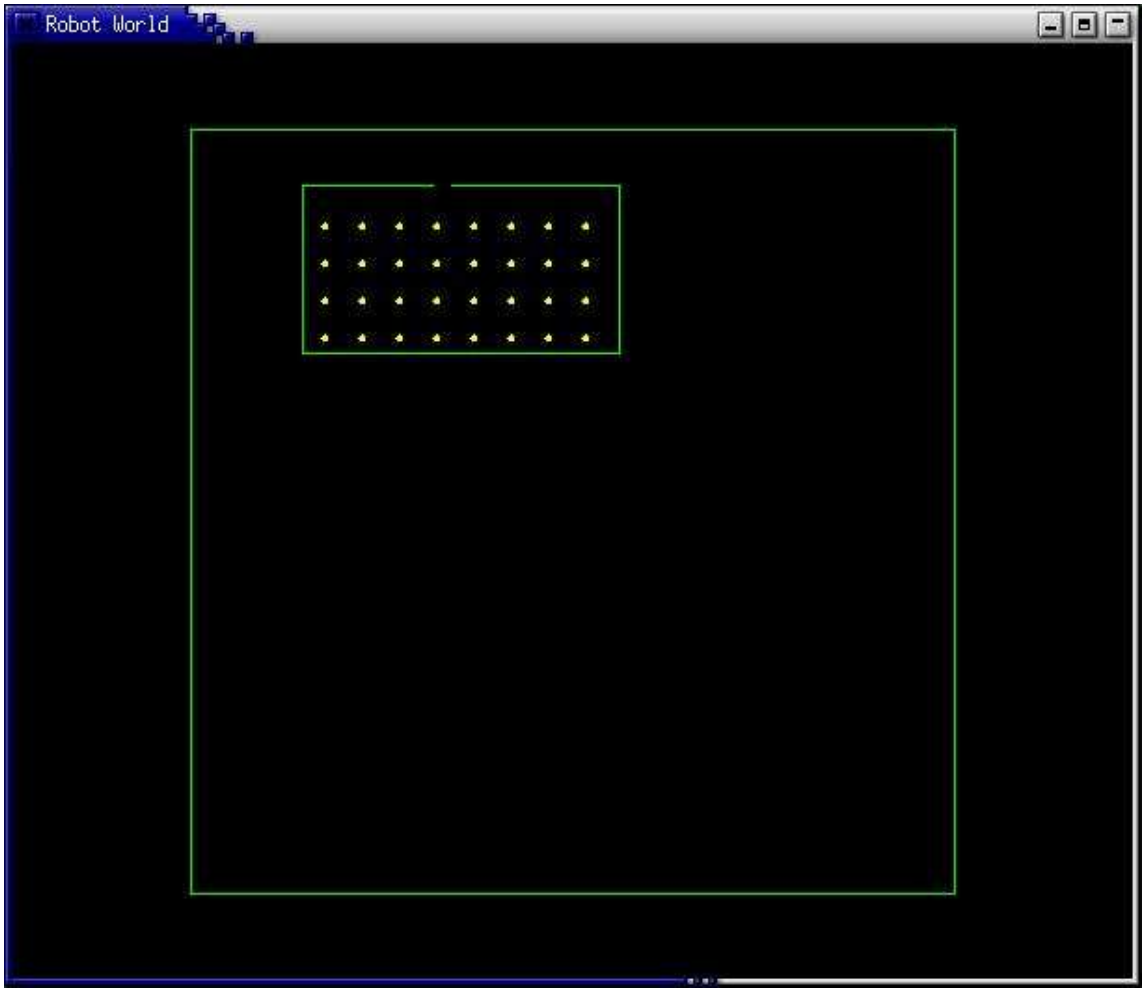
Hauptfunktion

```
spiral :: Robot ()
spiral = penDown >> loop 1 where
 loop n =
   let turnMove = do turnRight; for n move
   in  do for 2 turnMove
          cond_ (isnt blocked) (loop (n+1))

main :: IO ()
main = runRobot spiral s0 g0
```

- Zeigen.

Beispiel 2: eine etwas kompliziertere Welt



Zerlegung des Problems

- Ziel: in Welten wie dieser alle Münzen finden.
- Dazu: Zerlegung in Teilprobleme

1. Von Startposition in Spirale nach außen
2. Wenn Wand gefunden, Tür suchen
3. Wenn Tür gefunden, Raum betreten
4. Danach alle Münzen einsammeln

- Schritt 1: Variation der Spirale

```
treasureHunt :: Robot ()
treasureHunt = do
  penDown; loop 1
   where loop n =
           cond blocked findDoor $
             do turnRight; moven n
                cond blocked findDoor $
                  do turnRight
                     moven n; loop (n+1)
```

- Schritt 2: Tür suchen

```
findDoor :: Robot ()
findDoor = do
  turnLeft
  loop
   where loop = do
           wallFollowRight
           cond doorOnRight
             (do enterRoom; getGold)
             (do turnRight; move; loop)
```

- Hilfsfunktion 2.1: Wand folgen

```
wallFollowRight :: Robot ()
wallFollowRight =
  cond_ blockedRight $
    do move; wallFollowRight

blockedRight :: Robot Bool
blockedRight = do
  turnRight
  b <- blocked
  turnLeft
  return b
```

- Hilfsfunktion 2.2: Tür suchen, Umdrehen

```
doorOnRight :: Robot Bool
doorOnRight = do
  penUp; move
  b <- blockedRight
  turnAround; move; turnAround; penDown
  return b

turnAround :: Robot ()
turnAround = do turnRight; turnRight
```

- Schritt 3: Raum betreten

```
enterRoom :: Robot ()
enterRoom = do
  turnRight
  move
  turnLeft
  moveToWall
  turnAround

moveToWall :: Robot ()
moveToWall = while (isnt blocked)
                   move
```

- Schritt 4: Alle Münzen einsammeln

```
getGold :: Robot ()
getGold = do
  getCoinsToWall
  turnLeft; move; turnLeft
  getCoinsToWall
  turnRight
  cond_ (isnt blocked) $
    do move; turnRight; getGold
```

- Hilfsfunktion 4.1: Alle Münzen in einer Reihe einsammeln

```
getCoinsToWall :: Robot ()
getCoinsToWall = while (isnt blocked) $
                    do move; pickCoin
```

- Hauptfunktion:

```
main = runRobot treasureHunt s1 g3
```

- Zeigen!

Zusammenfassung

- Zustandsübergangsmonaden
  - Aktionen als Zustandsübergänge von `RealWorld#`
- Die Roboterkontrollsprache IRL
  - Einbettung einer imperativen Sprache in Haskell
- Abstrakter als direkte Modellierung (`Robot` vs. `Car`)