Bordeaux

Formale Entwicklung einer Rollstuhlstuhlsteuerung

Lutz Schröder, Till Mossakowski und Christoph Lüth

02.06.2004

Überblick

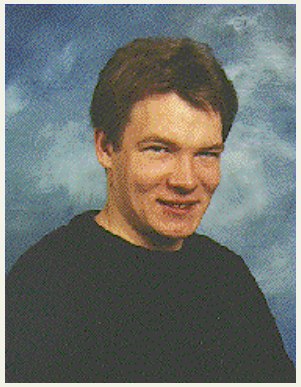
1. Organisatorisches
2. Der Rollstuhl Rolland
3. Die Imperative Roboterkontrollsprache IRL
4. Die Haskell Specification Language HasSLe
5. Schlußbemerkungen

Organisatorisches

Die Betreuer

Lutz Schröder

Till Mossakowski



Christoph Lüth

- Wissenschaftliche Assistenten AG Krieg-Brückner
  - Formale Methoden
  - Programmiersprachen
  - Kognitive Robotik (indirekt)

Das Projekt Bordeaux

Das Problem in der Robotik:

- Hardwarenahe Programmierung,
- Wenig Abstraktion,
- Korrektheitsbeweise schwer.

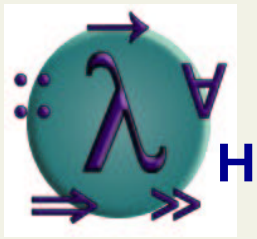
Das Projekt Bordeaux

Das Problem in der Robotik:

- Hardwarenahe Programmierung,
- Wenig Abstraktion,
- Korrektheitsbeweise schwer.

Vorgeschlagene Lösung:

- Nutzung funktionaler Sprachen;
- Entwicklung einer eingebetteten Sprache Bordeaux,
- basierend auf Haskell.

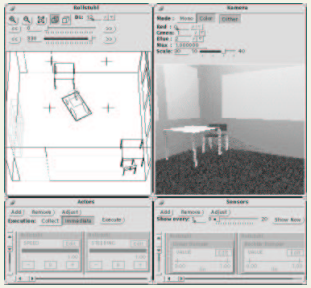


Haskell

Abstraktion

Algorithmen

Bordeaux



Rollstuhl

Simulator

Beweiser

Implementation

Verifikation

Projektplan 1. Semester – Vorschlag! –

- Einarbeitung & Zielfindung

Projektplan 1. Semester – Vorschlag! –

- Einarbeitung & Zielfindung
- Fortgeschrittene Funktionale Programmierung (Kurs):
  - Funktionale Robotik
  - Eingebette Sprachen (DSL)
- Grundlagen der Robotik (LV)
- Formale Spezifikation und Entwicklung (LV)

Projektplan 2. Semester – Vorschlag! –

- Technische Vorarbeiten

Projektplan 2. Semester – Vorschlag! –

- Technische Vorarbeiten
- Entwurf und Implementierung der DSL Bordeaux
- Formale Spezikation und Verifikation von funktionalen Programmen mit Isabelle/HOLCF und Hassle

Projektplan 3. Semester – Vorschlag! –

- Implementierung von BORDEAUX abschliessen
- Anschluss an Rolland
- Anschluss an Simulator
- Spezifikation und Beweis: erste Schritte

Projektplan 4. Semester – Vorschlag! –

- Beweise ausgewählter Eigenschaften
- Fortgeschrittene Algorithmen
  - Routengraph, Navigation, etc.
- Fröhliche Projektpräsentation
- Projektbericht

Begleitende Lehrveranstaltungen

- Techniken zur Entwicklung korrekter Software I
  (Mossakowski, Schröder)
- Fortgeschrittene Funktionale Programmierung (Lüth)
- (Kognitive) Robotik (Kirchner)
- Techniken zur Entwicklung korrekter Software II
  (Mossakowski, Schröder)
- Algebraische Spezifikation (NN)
- Logik (NN)

Projektausstattung und Benotung

- Projektraum, Projektrechner (FB3)
- Zugriff auf Rollstuhl, Laptop? (AG BKB)
- Klausurtagungen
- Benotung: Progressive Teilnoten pro Semester
  - Projektarbeit: implementierung, spezifizieren, etc;
  - Referate, mündliche Beteiligung im Plenum;
  - Hilfsarbeiten: Orga-Team, Sys-Admin, Doku, . . .

Der Rollstuhl Rolland

Der Bremer Autonome Rollstuhl Rolland

- Modell Champ, Fa. Meyra.
- Basisplattform
  DFG-Projekt SafeRobotics,
  SFB Raumkognition

Rolland

Aktorik:

- Ansteuerbar über USB $\longleftrightarrow$ seriell $\longleftrightarrow$ CAN-Bus
- Geschwindigkeit links/rechts; Hupe, Blinker, Licht.

Zusätzliche Sensorik:

- Laser-Scanner vorne/hinten, Odometrie

Programmierung des Rollstuhls

- Eingabe: Sensorik
  - Laser-Scanner (wo sind die Hindernisse?)
  - Odometrie (wo bin ich?)
- Ausgabe: Steuerkommandos
  - Geschwindigkeit
  - Lenkwinkel
- Ansteuerung: Kontrollschleife
  - Neue Steuerdaten alle 32 ms.

Haskell in Space II: Roboter

Inhalt

- Domain Specific Languages
- Modellierung von Zuständen: Monaden
- Beispiel für eine DSL: Roboterkontrollsprache IRL

Domain Specific Languages (DSL)

- DSL: Sprache für speziellen Problembereich
  - Im Gegensatz zu universalen Programmiersprachen
  - Beispiel: LaTeX, Shell-Skripte, Spreadsheets, . . .
- Implementierung von DSLs:
  - Einbettung in Haskell
- Beispiel: Imperative Roboterkontrollsprache IRL
- Dazu: imperative Konzepte in Haskell

Zustandsübergangsmonaden

- 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`.

- `ST` ist Instanz der Typklasse `Monad`:

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

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`

IRL im Beispiel

- Alle Roboterkommandos haben Typ `Robot a`
  - Bewegung `move :: Robot ()`, `turnLeft :: Robot ()`
  - Roboter kann zeichnen: `penUp :: Robot()`, `penDown :: Robot()`
  - Damit: 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
cond1 :: Robot Bool-> Robot ()-> Robot ()
while :: Robot Bool-> Robot ()-> Robot ()
```

- Bsp: Ausweichen

```
evade :: Robot ()
evade = do cond1 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
```

- `Robot a` transformiert Robotstate.
  - 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))
```

- Positionen:

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

- Richtungen:

```
data Direction = North | East | South | West
```

  - Hilfsfunktionen: Rechts-/Linksdrehungen:

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

- Die Welt:

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

  - Enthält für Feld `(x,y)` die Richtungen, in denen erreichbare Nachbarfelder sind.

Beispiel 1

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

- Hauptfunktion:

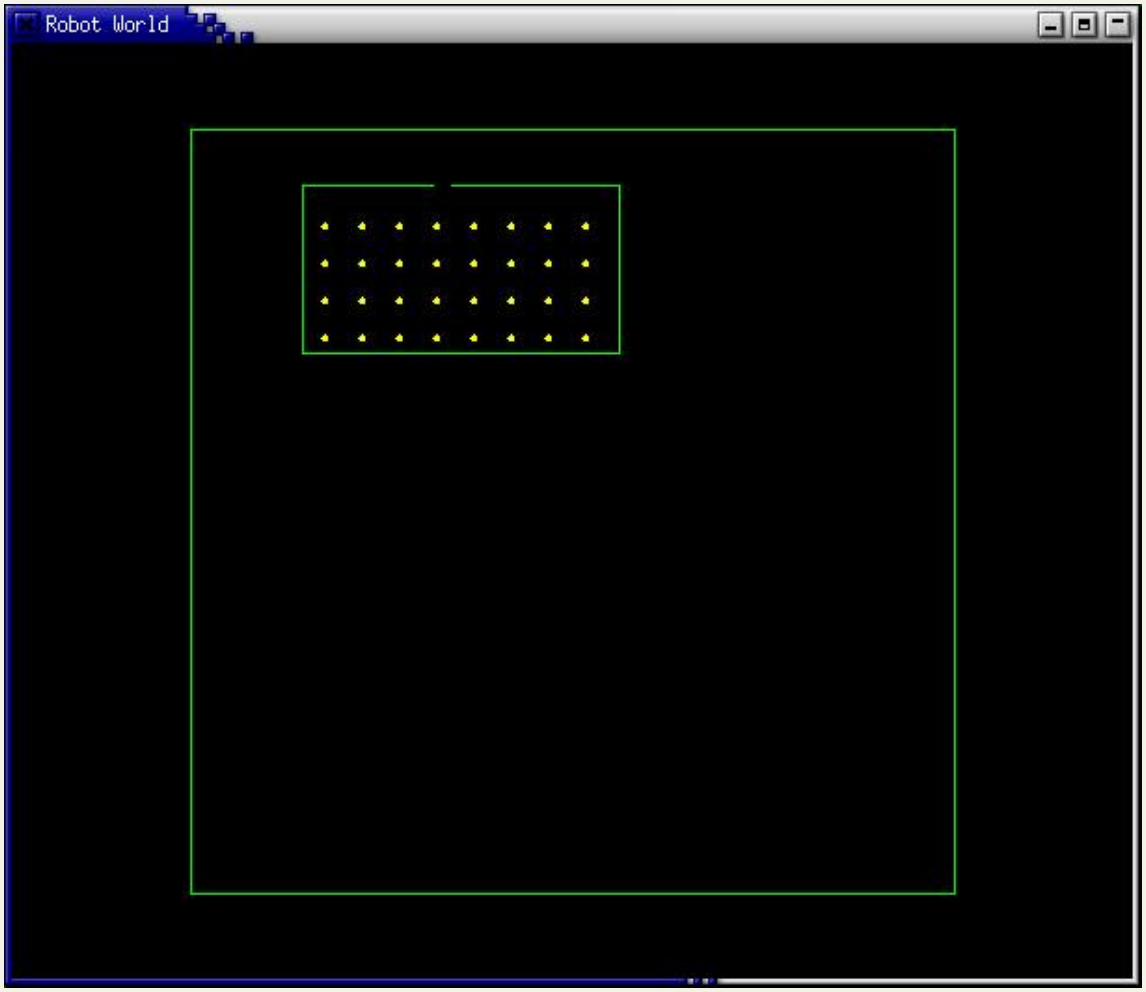
```
spiral :: Robot ()
spiral = penDown >> loop 1 where
 loop n =
   let turnMove = do turnRight; for n move
   in  do for 2 turnMove
          cond1 (isnt blocked) (loop (n+1))

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

- Zeigen!

Beispiel 2

Eine etwas komplizierte Welt:



- Ziel: in dieser Welt 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 =
  cond1 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
  cond1 (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

- Zustandstransformationen
  - Aktionen als Transformationen der `RealWorld`
- Die Roboterkontrollsprache IRL
  - Einbettung einer imperativen Sprache in Haskell
  - Der Robotersimulator
- Beispiel für eine domänenspezifische Sprache (DSL).
  - Hier in Haskell eingebettet.
  - Wegen flexibler Syntax, Typklassen und Funktionen höherer Ordnung gut möglich.

Die Haskell Specification Laguage (HasSLe)

Warum Spezifikation?

- Klärung der Anforderungen
- Entwicklung eines Designs als Design-Spezifiaktion, vor der eigentlichen Codierung
  $\Rightarrow$ ermöglicht, Entwicklung in einem frühen Stadium zu korrigieren
- Spezifikationen bringen auch konzeptionelle Klarheit
- "Vertrag" zwischen AuftraggberIN und EntwicklerIN
- Nachträgliche Analyse eines Programms

Warum formale Spezifikation?

- Programme sind fast immer fehlerhaft
  Das kann z.B. Personen gefährden (Auto, Rollstuhl)
- Informale Spezifikationen sind oft mehrdeutig
  Bsp.: Jan Bergstra schrieb 100 Programme, mit denen je zwei Java-Compiler unterschieden werden konnten
- Formale Spezifikationen haben eine eindeutige Semantik
- Formale Spezifikation auch nachträglich sinnvoll (Sicherheitseigenschaften)

Verwendung formaler Spezifikationen

- als präzise Dokumentation
- für systematisches Testen (in Haskell: QuickCheck)
- für automatisches Beweisen und Modelchecking
- für halb-automatisches, interaktives Beweisen

Erfolgsbeispiele formaler Spezifikation

- vollständige formale Verifikation der Pentium 4-Arithmetik (nach Pentuim bug)
- NASA benutzt Spezifikation von physikalischen Einheiten (nach Marssonden-Crash)
- Verifikation des Java Bytecode-Verifiers
- 12 Deadlocks in Occam-Code für internationale Raumstation gefunden

Die Haskell Specification Laguage (HasSLe)

- Annotation von Haskell-Programmen mit logischen Formeln
- Kleine Erweiterung der Haskell-Syntax
- Präzisere Beschreibung der Funktionalität direkt im Haskell-Code
- Mathematisch präzises Kriterium für formale Verifikation
- Effizientes Runtime-Debugging

Beispiel: Listen

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:l) = reverse l ++ [x]
{-# AXIOMS
  forall l -> length l = length(reverse l)
  forall l -> reverse(reverse(l)) = l
 #-}
```

Beispiel: Sortieren

```
sort :: Ord a => [a] -> [a]
sort = (?)
{-# AXIOMS
  "permutation" forall x::a l::[a] ->
       x `elem` l = x `elem` sort l
  "ordered" forall x,y::a l,l1,l2::[a] ->
      sort l = l1++[x,y]++l2 => x <= y
 #-}
```

Implementierung

```
insert :: Ord a => (a,[a]) -> [a]
insert(x,[]) = [x]
insert(x,y:l) = if x <= y then x:insert(y,l)
                          else y:insert(x,l)

sort :: Ord a => [a] -> [a]
sort([]) = []
sort(x:l) = insert(x,sort(l))
```

Beispiel: Pretty printing

```
(<>) :: Doc -> Doc -> Doc -- nebeneinander setzen
($$) :: Doc -> Doc -> Doc -- untereinander setzen
next :: Int -> Doc -> Doc -- um i Felder einr{\"u}c
```

Beispiel: Pretty printing

```
{-# AXIOMS
 forall x y z -> (x <> y) <> z = x <> (y <> z)
 forall x y z -> (x $$ y) $$ z = x $$ (y $$ z)
 forall x     -> x <> text "" = x
 forall k x y -> x <> next k y = x <> y
 forall k l x -> next k (nest l x) = nest (k+l) x
 forall x     -> next 0 x = x
 forall x y z -> (x $$ y) <> z = x $$ (y <> z)
 #-}
```

Beispiel: Rollstuhl

```
{-# AXIOMS
 "right_left"
   (do turnRight; turnRight; turnRight) = turnLeft
 "stops_at_wall" <move> true => isnt blocked
 #-}
```

Beispiel: Rollstuhl

```
{-# AXIOMS
 "right_left"
   (do turnRight; turnRight; turnRight) = turnLeft
 "stops_at_wall" <move> true => isnt blocked
 #-}
```

- Sicherheits-Eigenschaften

Beispiel: Rollstuhl

```
{-# AXIOMS
 "right_left"
   (do turnRight; turnRight; turnRight) = turnLeft
 "stops_at_wall" <move> true => isnt blocked
 #-}
```

- Sicherheits-Eigenschaften
- Liveness-Eigenschaften

Werkzeuge für HasSLe

- Haskell-Compiler/Interpreter zum Ausführen des Haskell-Programms
- QuickCheck zum Runtime-Testen der AXIOMS
- Isabelle/HOLCF zum Beweisen der AXIOMS

Übersetzung von <move>

```
<move> true => isnt blocked
```

wird codiert als

```
(forall q ->
do q; move; return false = do q; move; return true
 => do q; return false = do q; return true)
 => isnt blocked
```

Abschließende Bemerkungen

Zusammenfassung

Ziele:

- Programmierung eines Rollstuhls auf abstraktem Niveau
  - Entwurf und Implementierung einer eingebettenen Sprache
  - Anschluss an Rolland
  - Anschluss an Simulator
- Spezifikation und Verifikation ausgewählter Eigenschaften

Interessenten:

- funktionaler Programmierung
- Mischung aus konkreter Hardware und abstrakter Logik.

Warum Bordeaux?

- Zukunftsfähigkeit — Zusatzkenntnisse
  - "Softwareentwicklung für das 21. Jh."
  - Java alleine reicht nicht mehr;
- Korrektheit und Sicherheit immer wichtiger.
- Zunehmende Industrierelevanz formaler Methoden.
- Zukunftsfeld eingebettete Systeme.
- Mehr Informationen demnächst unter

  `http://www.informatik.uni-bremen.de/agbkb/lehre/bordeaux`