Modellierung von Call-Time Choice als Effekt unter Verwendung von Freien Monaden

Niels Bunkenburg

27. März 2018

Arbeitsgruppe für Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Motivation

```
let x = 0 ? 1 in x + x  ≠  (0 ? 1) + (0 ? 1)
```

- Ersetzungsregeln sind in Curry nicht immer wie in Haskell anwendbar
- Beweise über die Semantik eines Programms sind schwierig
- Ansatz: Übersetzung des Programms in andere Sprache
- Modellierung der Effekte (z.B. Nichtdeterminismus)

Programm und Effektsyntax

- Programm

```
data Prog sig a  = Return a
                 | Op (sig (Prog sig a))
```

- Effektsyntax

```
data ND p = Fail | Choice p p
```

- Nichtdeterministisches Programm **`Prog ND`** entspricht

```
data NDProg a = Return a
              | Fail
              | Choice (NDProg a) (NDProg a)
```

Handler

Effekte werden durch Handler verarbeitet

```
runND :: Prog ND a -> [a]
runND (Return a)   = [a]
runND Fail         = []
runND (Choice p q) = runND p ++ runND q
```

→ Der Handler bestimmt die Semantik eines Effekts

Funktionen

```
coin :: Prog ND Int
coin = Choice (return 0) (return 1)
```

Funktionen

```
coin :: Prog ND Int
coin = Choice (return 0) (return 1)

addM :: Prog sig Int -> Prog sig Int -> Prog sig Int
addM p1 p2 = do
  i1 <- p1
  i2 <- p2
  return $ i1 + i2
```

→ **liftM** bei strikten Funktionen

Funktionen

```
coin :: Prog ND Int
coin = Choice (return 0) (return 1)

addM :: Prog sig Int -> Prog sig Int -> Prog sig Int
addM p1 p2 = do
  i1 <- p1
  i2 <- p2
  return $ i1 + i2
```

→ **liftM** bei strikten Funktionen

```
orM :: Prog sig Bool -> Prog sig Bool -> Prog sig Bool
orM p1 p2 = p1 >>= \b -> case b of
                           True  -> return True
                           False -> p2
```

→ Pattern Matching erfordert Bind

Beispielausdrücke

```
λ> run $ addM (return 42) undefined
*** Exception: Prelude.undefined

λ> run $ orM (return True) undefined
True

λ> runND coin
[0,1]
```

Beispielausdrücke

```
λ> run $ addM (return 42) undefined
*** Exception: Prelude.undefined

λ> run $ orM (return True) undefined
True

λ> runND coin
[0,1]

λ> putStrLn . pretty . runND $ addM coin coin
?
|---- ?
       |---- 0
       |---- 1
|---- ?
       |---- 1
       |---- 2
```

Call-Time Choice

- Nichtdeterminismus und Sharing

```
Prelude> let x = 0 ? 1 in x + x
0
2
```

- `let` entspricht Sharing-Effekt

```
?1
|---- ?1
       |---- 0
       |---- 1
|---- ?1
       |---- 1
       |---- 2
```

→ Sharing-Effekt vergibt IDs für Choices

Sharing als Effekt mit Scope

```
data Share p = Share p
```

Sharing als Effekt mit Scope

```
data Share p = Share p
```

Gegenbeispiel

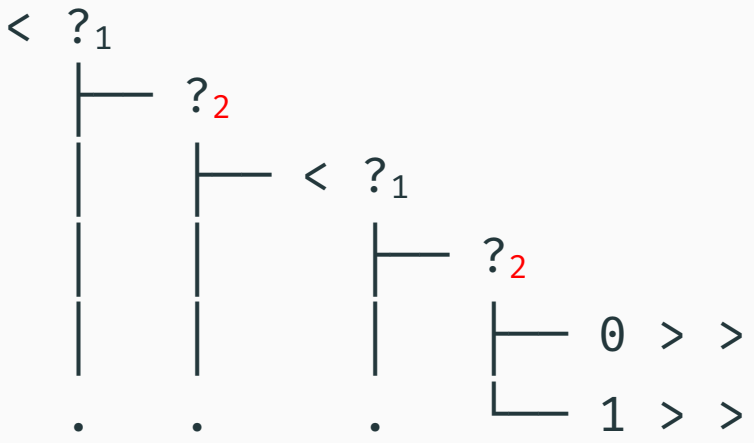
```
let x = coin in (x + coin) + (x + coin)
```

Sharing als Effekt mit Scope

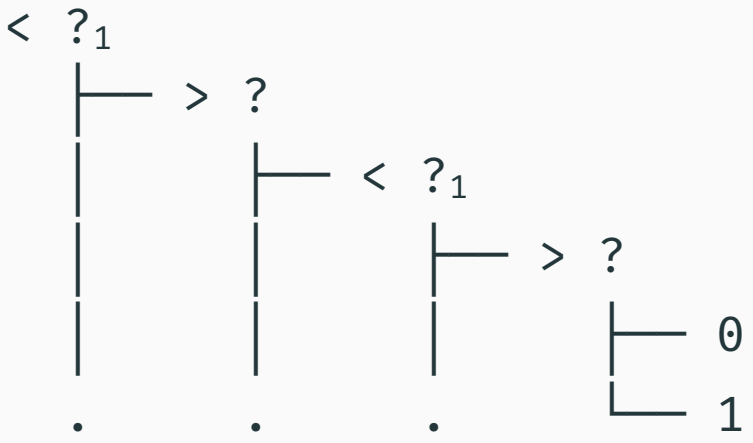
```
data Share p = Share p
```

Gegenbeispiel

```
let x = coin in (x + coin) + (x + coin)
```



Einfache Implementierung



Richtige Implementierung

Sharing-Effekt – Korrekte Implementierung

```
data Share p = BShare Int p | EShare Int p
```

Sharing-Effekt – Problem

```haskell
do fx <- share coin
   addM fx fx

do fx <- return $ do
     i <- get
     put (i + 1)
     share' i coin
   addM fx fx -- State Code wird dupliziert!
```

Sharing-Effekt – Problem

```haskell
do fx <- share coin
   addM fx fx
```

```haskell
do fx <- return $ do
     i <- get
     put (i + 1)
     share' i coin
   addM fx fx -- State Code wird dupliziert!
```

```haskell
do fx <- do -- State Code wird ausgewertet
     i <- get
     put (i + 1)
     return (share' i coin)
  addM fx fx
```

→ Zwei Programmebenen sind notwendig

Sharing-Effekt – Nested Sharing

Nested Sharing

```
let x = let y = coin
        in y + y
in x + x
```

```
share p = do
  i <- get
  put (i * 2)
  return . share' i $ do
    put (i * 2 + 1)
    p
```

→ Verschachtelte Aufrufe von **share** benötigen frische IDs

Sharing-Effekt – Deep Sharing

Deep Sharing

Geliftete Datentypen erlauben Effekte in den Komponenten,
z.B. `data List m a = Nil | Cons (m a) (m (List m a))`

```
let xs = [coin]
in (xs, xs)
```

```
share p = do
  i <- get
  put (i * 2)
  return . share' i $ do
    put (i * 2 + 1)
    x <- p
    shareArgs share x
```

→ Rekursive **share** Aufrufe für Komponenten notwendig

Zusammenfassung und Ausblick

Zusammenfassung

- Effekte können als Instanzen des Datentyps **`Prog`** modelliert werden
- Handler setzen Effekte im Programm um
- Call-Time Choice in Curry wird durch Nichtdeterminismus und Sharing modelliert
- Sharing kennzeichnet Choices geschickt mit IDs

Ausblick

- Drei Ansätze $\rightarrow$ Implementierung in Coq
- Beweisen der *laws of sharing* für die Implementierung
- Beweise über Eigenschaften von konkreten Curry Programmen