

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

`let x = 0 ? 1 in x + x` \neq `(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)
```

- **Prog** f ist eine Monade, wenn f ein Funktor ist

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

```
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 $ addM coin coin
```

```
?  
├─ ?  
│  └─ 0  
│    └─ 1  
└─ ?  
    └─ 1  
      └─ 2
```

Call-Time Choice

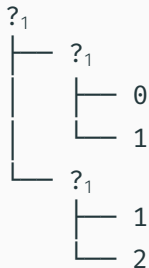
- Nichtdeterminismus und Sharing

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

0

2

- **let** entspricht Sharing-Effekt




→ 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:

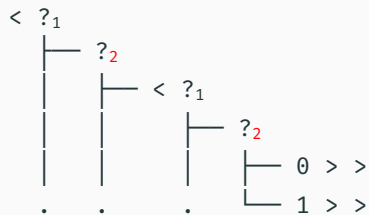
```
let x = coin in (x + coin) + (x + coin)
```

Sharing als Effekt mit Scope

data Share p = ~~Share~~ p

Gegenbeispiel:

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



Einfache Implementierung



Richtige Implementierung

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

→ Smartkonstruktor notwendig, um IDs zu generieren

Sharing-Effekt – Korrekte Implementierung

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

→ Smartkonstruktor notwendig, um IDs zu generieren

```
share :: Prog sig a -> Prog sig (Prog sig a)
```

```
share p = do
  i <- get
  put (i + 1)
  return $ do
    begin i
    x <- p
    end i
  return x
```

Sharing-Effekt – Nested Sharing

Nested Sharing

Sharing kann beliebig tief verschachtelt auftreten

Beispiel:

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

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

Deep Sharing

Geliftete Datentypen erlauben Effekte in den Komponenten

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

Beispiel:

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

→ Rekursive **share** Aufrufe für Komponenten notwendig

- Interaktiver Beweisassistent mit funktionaler Spezifikationssprache
- Alle Definition müssen nachweislich terminieren

Interessante Aspekte im Vergleich zu Haskell:

- Modellierung von Nicht-Striktheit in strikter Sprache
- Beweise über Curry Programme

Prog entspricht der Freien Monade **Free**

```
Inductive Free F A :=  
| pure : A -> Free F A  
| impure : F (Free F A) -> Free F A.
```



Non-strictly positive occurrence **of "Free" in**
"F (Free F A) -> Free F A".

→ Alternative Darstellung von Funktoren möglich?

Darstellung von Funktoren als Container

Container abstrahieren Datentypen, die (polymorphe) Werte enthalten

```
Class Container :=  
  {  
    Shape    : Type;  
    Pos      : Shape -> Type  
  }.
```

```
Inductive Ext (C : Container) A :=  
  ext : forall s : Shape,  
        (Pos s -> A) -> Ext C A.
```

Die **Erweiterung eines Containers** ist isomorph zum ursprünglichen Datentyp

Funktoren als Container: Choice

```
Inductive Choice (A : Type) :=  
| cfail    : Choice A  
| cchoice  : option ID -> A -> A -> Choice A.
```

```
Inductive ShapeChoice :=  
| sfail    : ShapeChoice  
| schoice  : option ID -> ShapeChoice.
```

```
Definition PosChoice (s: ShapeChoice) : Type :=  
  match s with  
  | sfail      => Void  
  | schoice _ => bool  
end.
```

Container Erweiterung für Choice

```
Instance CChoice : Container := {  
  Shape := ShapeChoice;  
  Pos   := PosChoice  
}.
```

```
Definition fromChoice A (z : Choice A) : Ext CChoice A :=  
  match z with  
  | cfail _ =>  
    ext sfail  
    (fun p : Void => match p with end)  
  | cchoice mid l r =>  
    ext (schoice mid)  
    (fun p : bool => if p then l else r)  
end.
```

- Darstellung aller Effekte und des Kombinationsfunktors als Container
- Statischer Effekt-Stack und Handlerreihenfolge
- Implementierung des **begin/end** Ansatzes für Sharing
- Problematisch: Handling von ungültigen **begin/end** Tags

→ Higher-Order Ansatz

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
- Coq: Darstellung von Funktoren als Container

Ausblick

- Ist es möglich, den Higher-Order Ansatz vollständig in Coq zu modellieren?
- Lassen sich andere Effekte, die von Sharing beeinflusst werden, ebenfalls darstellen?

Higher-Order Implementierung

```
data HShare m a = forall x. Share ID (m x) (x -> m a)
```

- Datentypen haben zusätzliches Argument `m`
- $(m \ a) \hat{=} p$ in bisheriger Implementierung
- Effekt Scope durch direkte Programmargumente
- Coq Implementierung: Indizierte Bi-Container
- Rekursive Datentypen nicht darstellbar

Sharing Operator

share :: Prog sig a -> Prog sig (Prog sig a)

```
share p = do
  (i, j) <- get
  put (i + 1, j)
  return $ do
    begin (i,j)
    put (i, j + 1)
    x <- p
    x' <- shareArgs share x
    end (i,j)
  return x'
```