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

Niels Bunkenburg

19. Dezember 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)

Grundlagen

- Programme

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

- Programmsignaturen

```
data (sig1 + sig2) p = Inl (sig1 p)
                     | Inr (sig2 p)
```

Beispieleffekte

- Effektfreie Programme

```
data Void p
data VoidProg a = Return a
```

- Nichtdeterministische Programme

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

→ Der konkrete Datentyp bestimmt die Syntax eines Effekts

Handler

Effekte werden durch Handler verarbeitet

```
run :: Prog Void a -> a
run (Return x) = x

runND :: Prog ND a -> Tree a
runND (Return a) = Leaf a
runND Fail       = Empty
runND (Choice p q) =
  let pt = runND p
      qt = runND q
  in Branch pt qt
```

→ 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
Choice (Leaf 0) (Leaf 1)
```

Beispielausdrücke

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

λ> run $ orM (return True) undefined
True

λ> runND coin
Choice (Leaf 0) (Leaf 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-Effekt – Idee

- Explizite Sharing Syntax

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

- ```
  let x = coin in x + x
  ```

  wird zu

  ```
  share coin >>= \fx -> addM fx fx
  ```

Wie werden Choice IDs innerhalb von **share** vergeben?
  ```

Sharing-Effekt – Datentyp und Handler

```
data Share p = Share p

runShare :: Prog (Share + ND) a -> Prog ND a
runShare (Return a) = return a
runShare (Share  p) = nameChoices p
runShare (Other op) = Op (fmap runShare op)

nameChoices :: Prog (Share + ND) a -> Prog ND a
nameChoices (Return a)     = return a
nameChoices (Share  p)     = nameChoices p
nameChoices Fail           = fail
nameChoices (Choice _ p q) =
  choice 42 (nameChoices p) (nameChoices q)
```

→ **Share** benötigt eine ID

Sharing-Effekt – IDs

```haskell
data Share p = Share Int p
```

→ **share** muss ID selbst generieren

Idee: Individuelle IDs durch State Effekt

```haskell
share p = return $ do
  i <- get
  put (i + 1)
  share' i p -- Smartkonstruktor für Share
```

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

```
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!
```

```
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

Verschachteltes Sharing

```
share (share coin >>= \fx -> addM fx fx)
  >>= \fy -> addM fy fy
```

```
?2
|-- ?2
    |-- ?4
        |-- ?4
            |-- 0
            |-- 1
        |-- ?4
            |-- 1
            |-- 2
            ...
```

Ohne Berücksichtigung von verschachteltem Sharing

```
?3
|-- ?3
    |-- ?3
        |-- ?3
            |-- 0
            |-- 1
        |-- ?3
            |-- 1
            |-- 2
            ...
```

Mit Berücksichtigung von verschachteltem Sharing

Sharing-Effekt – Vollständiger Handler

```haskell
runShare :: (Functor sig, ND <: sig)
  => Prog (Share + sig) a -> (Prog sig a)
runShare (Return a)  = return a
runShare (Share i p) = nameChoices i 1 p
runShare (Other op)  = Op (fmap runShare op)

nameChoices :: (ND <: sig)
  => Int -> Int -> Prog (Share + sig) a -> Prog sig a
nameChoices scope next prog = case prog of
  Return a  -> Return a
  Share i p -> nameChoices i 1 p
  Choice _ p q ->
    let p' = nameChoices scope (2 * next)      p
        q' = nameChoices scope (2 * next + 1)  q
    in choiceID (Just (scope, next)) p' q'
  Other op -> Op (fmap (nameChoices scope next) op)
```

Laws of Sharing

Purely Functional Lazy Non-deterministic Programming

$$\mathsf{ret}\, x \gg\!= k = kx \qquad \text{(Lret)}$$

$$a \gg\!= \mathsf{ret} = a \qquad \text{(Rret)}$$

$$(a \gg\!= k_1) \gg\!= k_2 = a \gg\!= \lambda x.\, k_1 x \gg\!= k2 \qquad \text{(Bassc)}$$

$$\emptyset \gg\!= k = \emptyset \qquad \text{(Lzero)}$$

$$(a \oplus b) \gg\!= k = (a \gg\!= k) \oplus (b \gg\!= k) \qquad \text{(Ldistr)}$$

$$\mathsf{share}\,(a \oplus b) = \mathsf{share}\, a \oplus \mathsf{share}\, b \qquad \text{(Choice)}$$

$$\mathsf{share}\,\emptyset = \mathsf{ret}\,\emptyset \qquad \text{(Fail)}$$

$$\mathsf{share}\,\bot = \mathsf{ret}\,\bot \qquad \text{(Bot)}$$

$$\mathsf{share}\,(\mathsf{ret}\,(c\, x_1 \ldots x_n)) = \mathsf{share}\, x_1 \gg\!= \lambda y_1 \ldots. \qquad \text{(HNF)}$$
$$\mathsf{share}\, x_n \gg\!= \lambda y_n.\, \mathsf{ret}\,(\mathsf{ret}\,(c\, y_1 \ldots y_n))$$

where c is a constructor with n non-deterministic components

Figure 1. The laws of a monad with non-determinism and sharing

Bäume

```
data Decision = L | R
type Memo = Map.Map ID Decision

dfs :: Memo -> Tree a -> [a]
dfs mem Failed = []
dfs mem (Leaf x) = [x]
dfs mem (Choice Nothing t1 t2) = dfs mem t1
                               ++ dfs mem t2
dfs mem (Choice (Just n) t1 t2) =
    case Map.lookup n mem of
      Nothing -> dfs (Map.insert n L mem) t1
              ++ dfs (Map.insert n R mem) t2
      Just L -> dfs mem t1
      Just R -> dfs mem t2
```

Explizite Scopes

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

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