Programmierung und Modellierung mit Haskell

Ausblicke

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

10. Juli 2014

Überschatten

```
schatten1  =  let x  = 1 in
              let y1 = x in
              let x  = 2 in
              let y2 = x in
              let x  = 3 in
              [x,y1,x,y2,x]
```

Was kommt heraus?

Überschatten

```
schatten1  =  let x  = 1 in
              let y1 = x in
              let x  = 2 in
              let y2 = x in
              let x  = 3 in
              [x,y1,x,y2,x]
```

Was kommt heraus? `[3,1,3,2,3]`

- Der Wert einer Variable/Bezeichners wird nie verändert.

  ⇒ Referentielle Transparenz
- Die `let`-Definition führt einen *neuen* lokale Abkürzung ein.
- Ein eventuell bereits vorhandener Bezeichner mit gleichem Namen wird **überschattet**, d.h. existiert weiterhin unverändert, ist aber nicht mehr ansprechbar.

Überschatten

Eine Überschattung kann auch nur vorübergehend stattfinden:

```
schatten2 = let x = 1          -- 1. Zeile
                y = let x = 2  -- 2. Zeile
                    in x       -- 3. Zeile
                z = x          -- 4. Zeile
            in [x,y,z]

schatten2 == [1,2,1]
```

Definition von `x` in Zeile 2 überschattet die Definition von `x` innerhalb des gesamten `let`-Ausdrucks (Zeile 2–3).
Außerhalb dieses `let`-Ausdrucks gilt die Definition von `x` in Zeile 1.
Diese wird z.B. dann wieder in Zeile 4 verwendet.

Keine Überschattung

Da eine Bezeichner für den gesamten Ausdruck gilt, darf man innerhalb eines einzigen `let`-Ausdruck den gleichen Bezeichner nur einmal definieren:

```
schatten3 = let x  = 1        -- 1. Zeile
                y1 = x        -- 2.
                x  = 2        -- 3.
                y2 = x        -- 4.
                x  = 3        -- 5.
            in [x,y1,x,y2,x] -- 6.
> :r
Code15.hs:1:17:
  Conflicting definitions for `x'
  Bound at: Code15.hs:1:17
            Code15.hs:3:17
            Code15.hs:5:17
Failed, modules loaded: none.
```

Keine Überschattung

```
schatten4 = let x  = 1        -- 1. Zeile
                y  = x        -- 2.
            in                -- 3.
            let x  = x + 1    -- 4.
                z  = x        -- 5.
                os = 1 : os   -- 6.
            in [y1,y2,x]      -- 7.
> shatten4
[1,
```

`let`-Definitionen sind in Haskell immer rekursiv. In Zeile 4 wird daher *nicht* der in Zeile 1 definierten Wert von `x` verwendet, sondern rekursiv der gerade definierte. Die Auswertung von Zeile 4 führt daher zu einem nicht-terminierenden Programm!
Zeile 6 demonstriert eine sinnvolle Verwendung des rekursiven `let`. Hauptverwendung ist jedoch die Definition von rekursiven Funktionen in `let`-Ausdrücken.

Überschattung überall

Überschattung kann in jedem Ausdruck stattfinden, in dem Bezeichner eingeführt werden. Neben `let` also auch in `where`, `case`, `do`, Funktionsdefinitionen, etc.

```
main = do let x = 1
          print x
          let p1 = print x
          let x = 1 + 1
          print x
          let p2 = print x
          let x = 1 + 1 + 1
          print x
          let p3 = print x
          p1; p2; p3; p1; p2; p3
```

Gibt nacheinander Zeilen mit den Werten $1, 2, 3, 1, 2, 3, 1, 2, 3$ aus.

Mehrkernprozessoren nutzen

- Viele Maschinen verfügen heute über mehrere Kerne, die Berechnungen ausführen können.
- Gleichzeitige Ausnutzung mehrerer Kerne ist jedoch schwierig, z.B. **Deadlocks** und **Race Conditions** drohen.
  Da die Anzahl der möglichen Programmabläufe exponentiell mit den Kernen wächst, wird das Testen eines Programms dadurch sehr schwierig.
- Aufgrund der referentiellen Transparenz bietet Haskell hier einen Ausweg an: So ist es zum Beispiel leicht möglich, ein Programm zu parallelisieren und dabei Deadlocks und Race Conditions a priori auszuschließen.

Beispiel: Parallelisierung

Einfache Parallelisierung mit Par-Monade:

```
import Control.Monad.Par   -->  cabal install monad-par
foo = runPar $ do
        x <- newFull left
        y <- newFull right
        resx <- get x
        resy <- get y
        return (x,y)
where
  left        = ..Ausdruck mit aufwändiger Auswertung..
  right       = ..Ausdruck mit aufwändiger Auswertung..
```

- Nur für Beschleunigung, garantiert aber Erhalt der Semantik
- Echte Nebenläufigkeit z.B. über State Transactional Memory

Beispiel: Nebenläufigkeit

```
import Control.Concurrent

sillyA = putStrLn $ "SillyA " ++ (show $ fib 45)
sillyB = putStrLn $ "SillyB " ++ (show $ fib 36)
sillyC = putStrLn $ "SillyC " ++ (show $ fib 27)

main = do putStrLn "Creating Threads."
          forkIO sillyA
          forkIO sillyB
          forkIO sillyC
          putStrLn "Waiting for completion."
          threadDelay 15000000
          putStrLn "Done."
```

- Reihenfolge der Ausgabe kann nicht vorhergesagt werden; nützlich für gleichzeitige Interaktion mit mehreren Dingen

Meta Programmierung

Viele Programmierer nutzen Präprozessoren, um Ihren Code für verschiedene Zwecke anzupassen.
Zum Beispiel:

```
<code>
#ifdef <spezielle version>
  <code>
#endif
<code>
```

Metaprogrammierung bedeutet, diesen Schritt konsequent weiter zu gehen: Ein Programm, dessen Ausführung ein Programm erzeugt, welches dann verwendet wird.

Template-Haskell ermöglicht die typ-sichere Programmierung verschiedener Sprachen mit Haskell, z.B. Javascript, HTML oder eben auch Haskell selbst

Beispiel: Metaprogrammierung in Haskell

Generische Projektion `$(projN 5 2) (1,2,3,4,5) == 2`

```
{-# LANGUAGE TemplateHaskell #-}
projNI :: Int -> Int -> ExpQ
projNI n i = lamE [pat] rhs
  where pat = tupP (map varP xs)
        rhs = varE (xs !! (i - 1))
        xs  = [ mkName $ "x" ++ show j | j <- [1..n] ]
```

Dynamische Erzeugung von Javascript in einem Haskell-Webserver:

```
...
forM_ heroList (\i -> toWidget [julius|
  function Show#{rawJS $ show i}Skill () {
    document.getElementById('heroImage').setAttribute('class','#{rawJS $ displayHero i}');
    document.getElementById('SkillAtag').innerHTML = '#{rawJS $ displayHeroSkill i 1}';
    document.getElementById('SkillBtag').innerHTML = '#{rawJS $ displayHeroSkill i 2}';
  |]
...
```

Vorlesung: FFP

Vorlesung: Fortgeschrittene Funktionale Programmierung
angeboten im kommenden Wintersemester (2+2) (6 ECTS)
Vorlesung: Freitags 10–12, Übung Mittwochs: 18–20
unter "Vertiefende Themen Bachelor/Master"

Mögliche Inhalte

- Testen mit QuickCheck
- Typsystem Erweiterungen
- Parallele und Nebenläufige Programme
- Template Haskell
- Webapplikationen
- Grafische Benutzeroberflächen

Prüfung erfolgt idealerweise über kleines Programmierprojekt